Titre: Title:	Analyse de variations de performance par comparaison de traces d'exécution
Auteur: Author:	François Pierre Doray
Date:	2015
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Pierre Doray, F. (2015). Analyse de variations de performance par comparaison de traces d'exécution [Master's thesis, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/1861/

UNIVERSITÉ DE MONTRÉAL

ANALYSE DE VARIATIONS DE PERFORMANCE PAR COMPARAISON DE TRACES D'EXÉCUTION

FRANÇOIS PIERRE DORAY
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
JUILLET 2015

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

ANALYSE DE VARIATIONS DE PERFORMANCE PAR COMPARAISON DE TRACES D'EXÉCUTION

présenté par : PIERRE DORAY François
en vue de l'obtention du diplôme de : Maîtrise ès sciences appliquées
a été dûment accepté par le jury d'examen constitué de :

M. MERLO Ettore, Ph. D., président
M. DAGENAIS Michel, Ph. D., membre et directeur de recherche
M. BELTRAME Giovanni, Ph. D., membre

DÉDICACE

À tous ceux qui me
rendent heureux
chaque jour.

REMERCIEMENTS

Je remercie d'abord mon directeur de recherche, Michel Dagenais, d'avoir cru en moi. Il a réussi à me transmettre sa passion pour Linux et m'a fait sentir à ma place au laboratoire de recherche sur les systèmes répartis ouverts et très disponibles (DORSAL). Il a aussi été très disponible tout au long de ma maîtrise.

Je remercie le Conseil de recherches en sciences naturelles et en génie du Canada (CRSNG) de m'avoir accordé une bourse d'études supérieures.

Je remercie Francis, Julien et Suchakra de m'avoir donné les pistes nécessaires pour aller jouer dans les entrailles du système d'exploitation Linux. J'ai aussi beaucoup aimé discuter avec eux de mes travaux et écouter leurs précieux conseils. Je remercie Simon et Mathieu, des amis sur lesquels j'ai toujours pu compter depuis le début du baccalauréat.

Je remercie Etienne Bergeron de m'avoir encouragé à me dépasser tout en m'amusant. Je remercie Mathieu Desnoyers de m'avoir suggéré des pistes très intéressantes lors de la réalisation de ce projet. J'ai beaucoup aimé que tous les deux me partagent leurs innombrables connaissances.

Je remercie mes parents qui m'ont encouragé dans tous les projets que j'ai entrepris dans ma vie. Je remercie mon frère Étienne qui a accepté toutes les demandes que je lui ai faites durant ce projet et avec qui je peux parler de bits. Je remercie Emmanuelle et Élisabeth d'être des sœurs que j'admire.

RÉSUMÉ

La performance est un requis très important pour bon nombre d'applications. Malheureusement, plusieurs facteurs affectent cette performance : contention pour l'accès à une ressource, mauvais algorithmes de synchronisation, attente de données en provenance du disque, etc. En raison de la taille du code source de certaines applications et des multiples niveaux d'abstraction entre le code applicatif et le matériel, plusieurs développeurs ne soupçonnent pas l'existence de toutes ces sources de latence.

Le traçage est une technique qui consiste à enregistrer des événements qui surviennent dans un système informatique. Un grand nombre de problèmes de performance peuvent être diagnostiqués à partir de l'information contenue dans une trace d'exécution. En raison de leur faible surcoût, les traceurs modernes peuvent être activés en continu sur des systèmes en production pour capturer des problèmes survenant rarement. Des outils spécialisés facilitent la navigation à travers le grand nombre d'événements contenus une trace d'exécution. Malheureusement, même avec ces outils, il peut être difficile de juger si le comportement observé est celui attendu sans une connaissance exhaustive du système analysé.

L'objectif de ce travail de recherche est de vérifier si le diagnostic de variations de performance peut être facilité par un algorithme qui identifie automatiquement les différences entre deux groupes de traces d'exécution. L'algorithme doit mettre en évidence les latences présentes dans un groupe d'exécutions anormalement longues, mais pas dans un groupe d'exécutions normales. Un développeur peut alors tenter d'éliminer ces latences.

Pour ce faire, nous introduisons d'abord deux nouveaux événements pour le traceur LTTng. L'événement `cpu_stack` rapporte périodiquement la pile d'appels des fils d'exécution utilisant le processeur. L'événement `syscall_stack` rapporte la pile d'appels des longs appels système. Ces deux événements nous permettent de déterminer si des latences présentes dans des traces d'exécution différentes ont été causées par le même code applicatif. Le traçage de l'événement `syscall_stack` a un surcoût moindre que celui du traçage traditionnel des appels système.

Nous proposons aussi une nouvelle structure de données appelée *enhanced calling context tree* (ECCT) pour représenter les latences affectant le temps de complétion d'une tâche. Ces latences peuvent être au niveau du matériel, du système d'exploitation ou du code applicatif. En présence d'interactions entre plusieurs fils d'exécution, nous utilisons un algorithme de calcul du chemin critique pour inclure dans le ECCT les latences introduites par chacun de ces fils. Nous utilisons les ECCTs à la fois pour stocker des métriques de performance de façon compacte et comme entrée de notre algorithme de comparaison.

Ensuite, nous présentons une interface graphique permettant de visualiser les différences entre des groupes d'exécution. Les groupes à comparer sont définis par l'utilisateur à l'aide de filtres. Les différences sont montrées à l'aide d'un outil de visualisation appelé *flame graph* différentiels ainsi que d'histogrammes. Les vues sont rafraîchies rapidement lorsque les filtres sont modifiés grâce à un algorithme de *map-reduce*.

L'efficacité de notre solution pour détecter, diagnostiquer et corriger des problèmes de performance majeurs est démontrée grâce à quatre études de cas menées sur des logiciels libres et d'entreprises. Nous mesurons aussi le surcoût du traçage des événements requis par notre analyse sur différents types d'applications et concluons qu'il est toujours entre 0.2% et 9%. Enfin, nous démontrons que notre solution développée pour Linux peut être adaptée pour fonctionner sur les systèmes d'exploitation Windows et Mac OS X.

ABSTRACT

Performance is a critical requirement for many applications. Unfortunately, many factors can affect performance: resource contention, bad synchronization algorithms, slow disk operations, etc. Because of the large codebase of some applications and because of the multiple levels of abstraction between application code and hardware, many developers are not aware of the existence of these sources of latency.

Tracing is a technique that consists of recording events that occur in a system. Many performance problems can be diagnosed using the information contained in an execution trace. Popular tracers achieve low overhead, which allows them to be enabled on production systems to capture bugs that occur infrequently. Specialized tools allow efficient navigation in the large number of events contained in execution traces. However, even with these tools, it is hard to determine whether the observed behavior is normal without a deep knowledge of the analyzed system.

The objective of this research project is to verify whether we can facilitate the diagnosis of performance variations with an algorithm that automatically identifies differences between two groups of execution traces. The algorithm must highlight delays that appear in a group of slow executions, but not in a group of normal executions. A developer can then try to eliminate these delays.

First, we introduce two new events for the LTTng tracer. The `cpu_stack` event periodically reports the call stack of threads running on the CPU. The `syscall_stack` reports the call stack of long system calls. Call stacks allow us to determine whether delays that appear in different execution traces are caused by the same application code. The overhead of tracing the `syscall_stack` event is less than that of traditional tracing of system calls.

Second, we propose a new data structure called enhanced calling context tree (ECCT) to describe delays that affect the completion time of a task execution. These delays can be at the hardware, operating system or application levels. When a task execution involves interactions between several threads, a critical path algorithm is used to include delays caused by each of them in the ECCT. We use ECCTs to store performance metrics compactly and as an input to our comparison algorithm.

Third, we present a GUI that shows differences between groups of executions. The user defines two groups of executions to compare using filters. Differences are shown using a visualization tool called differential flame graph along with histograms. A map-reduce algorithm

is used to quickly refresh the views when filters are modified.

We present four case studies, carried on open-source and enterprise software, to demonstrate that our solution helps to detect, diagnose and fix major performance problems. We measure the overhead of tracing the events required by our analysis on different kinds of applications and conclude that it is always between 0.2% and 9%. Finally, we show that our solution, developped for Linux, can be adapted for the Windows and Mac OS X operating systems.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES SIGLES ET ABRÉVIATIONS

API	Application Programming Interface
CCT	Calling Context Tree
CPU	Central Processing Unit
CTF	Common Trace Format
ECCT	Enhanced Calling Context Tree
ETW	Event Tracing for Windows
ELF	Executable and Linkable Format
LTTng	Linux Trace Toolkit Next Generation
RCU	Read-Copy Update
RPC	Remote Procedure Call

LISTE DES ANNEXES

CHAPITRE 1 INTRODUCTION

L'omniprésence des processeurs multicœurs, la montée en popularité de l'infonuagique et l'arrivée de nouveaux langages de programmation dynamiques favorisent le développement d'applications puissantes, mais aussi complexes. Les développeurs doivent être à l'affut de plusieurs problèmes propres aux systèmes parallèles : concurrence critique, interblocage, contention pour l'accès à une ressource... Cette tâche peut toutefois être complexe en raison des multiples niveaux d'abstraction qui séparent le code applicatif du matériel. De plus, ces problèmes sont difficiles, voire impossibles à diagnostiquer avec les débogueurs et profileurs traditionnels.

Le traçage est une technique efficace pour collecter de l'information sur les problèmes affectant les systèmes parallèles. Il est toutefois ardu d'analyser le grand volume d'information présent dans une trace d'exécution sans des outils appropriés.

Ce mémoire propose d'avoir recours à la comparaison de traces d'exécution pour identifier automatiquement la cause d'une variation de performance dans un système parallèle. Nous tentons de démontrer que cette technique permet à un développeur de détecter, diagnostiquer et corriger une grande variété de problèmes de performance sans posséder une connaissance approfondie du système étudié.

1.1 Définitions et concepts de base

1.1.1 Tâche et exécution

Une tâche est un travail pouvant être accompli par un système informatique. Par exemple :

- Retourner les résultats d'une recherche sur le Web.
- Ajouter un utilisateur dans une base de données.
- Effectuer le rendu d'une image d'un jeu en 3D.

Une exécution est une instance d'une tâche. On peut associer des propriétés à une exécution : estampille de temps, durée totale, nombre de fautes de page générées... L'exécution d'une tâche complexe peut requérir des interactions entre plusieurs composants répartis sur plusieurs machines.

Ce travail de recherche s'intéresse aux différences de performance qui surviennent entre plusieurs exécutions d'une même tâche.

1.1.2 Traçage

Un traceur est un outil capable d'enregistrer des événements survenant dans un système informatique. Un événement est constitué d'un type, d'une estampille de temps et de données utiles. Un événement est généré lorsqu'un programme rencontre un point de trace. Un point de trace peut être inséré statiquement ou dynamiquement dans une application en espace utilisateur ou dans le noyau du système d'exploitation.

Une trace de l'espace utilisateur offre une vue du comportement d'un système proche du code applicatif. Pour générer une telle trace, il faut instrumenter chaque application. Une trace du noyau offre une vue de plus bas niveau. Elle peut révéler toutes les interactions entre les applications et le système d'exploitation. Les auteurs de [1] dénotent qu'une grande proportion des problèmes de performance sont liés aux interactions avec le système d'exploitation.

La journalisation peut être considérée comme une forme de traçage. Cependant, quand on parle de traçage, on fait généralement référence à l'enregistrement d'événements à un débit beaucoup plus élevé. Les changements de contexte, la réception de paquets réseau ou l'occurrence de fautes de pages sont des exemples d'événements pertinents à tracer pour diagnostiquer un problème de performance.

Une force des principaux traceurs est qu'ils ont un impact minime sur le comportement des systèmes sur lesquels ils sont utilisés. Cette caractéristique fait d'eux des outils de choix pour déboguer des erreurs de synchronisation subtiles, où la moindre perturbation du système peut empêcher le comportement fautif de se reproduire. Les traceurs sont aussi fort utiles pour diagnostiquer des erreurs intermittentes qui se produisent uniquement en production. Grâce à leur faible surcoût, ils peuvent surveiller un système en production sur une longue période de temps, jusqu'à ce que l'erreur recherchée se manifeste.

Tout comme les profileurs, les traceurs peuvent révéler quelles fonctions d'une application consomment du temps processeur, de la mémoire, des accès réseau, etc. Cependant, ils ne se contentent pas de produire des valeurs totales pour un intervalle de temps donné : ils fournissent des détails pour chaque occurrence d'un événement. Cela permet par exemple de repérer des opérations dont la distribution des temps d'exécution est multimodale. Il est aussi possible de savoir ce qui se passe sur l'ensemble du système lorsqu'un délai plus long que l'ordinaire survient.

1.1.3 État d'un système

L'état d'un système peut être représenté par un ensemble d'attributs associés à des valeurs. Le tableau 1.1 montre l'état d'un système où le processeur 1 traite une interruption, le

processeur 2 exécute le fil d'exécution 7 et le descripteur de fichier 6 du fil d'exécution 7 réfère à `Fichier.txt`. La section 2.2.1 explique comment construire un modèle permettant de suivre l'état d'un système tout au long de la lecture d'une trace.

Tableau 1.1 État d'un système

Attribut	Valeur
Processeurs / 1 / État	Interruption liée au disque
Processeurs / 2 / État	Exécution
Processeurs / 2 / Fil d'exécution	7
Fils d'exécution / 7 / Descripteurs de fichiers / 6	`Fichier.txt`

Suivre l'état courant d'un système tout au long de la lecture d'une trace permet de donner un sens supplémentaire aux événements. Par exemple, l'événement `sched_wakeup [cpu_id=1, target_pid=42]` est généré lorsque le processeur 1 réveille le fil d'exécution 42. Dans l'état courant présenté au tableau 1.1, le processeur 1 est en train de traiter une interruption liée au disque. En combinant les données l'événement `sched_wakeup` et de l'état courant, on peut déduire qu'une requête au disque faite par le fil d'exécution 42 vient de se terminer.

1.2 Éléments de la problématique

Les applications développées aujourd'hui sont à la fois puissantes et complexes. Le moteur de recherche Google a recours à des milliers de machines pour retourner les résultats les plus pertinents à chaque requête qu'il reçoit en une fraction de seconde [2]. Netflix utilise la virtualisation pour ajuster dynamiquement les ressources de son infrastructure à la demande des utilisateurs [3]. Les téléphones intelligents possèdent des processeurs multicœurs leur permettant d'être à la fois plus performants et moins énergivores. Les langages dynamiques (Java, Javascript, Python...) simplifient le développement en offrant entre autres des librairies polyvalentes, une gestion automatique de la mémoire et des binaires portables. Dans un tel contexte, comprendre l'ensemble des facteurs pouvant affecter la performance d'une application peut représenter un défi de taille pour un seul ingénieur.

La performance est toutefois extrêmement importante pour un grand nombre d'applications. Les longs temps de réponse sont parmi les principales causes de frustration des utilisateurs identifiées par [4]. Le taux de conversion d'un site de commerce électronique peut être réduit de 7% par une latence de 1 seconde [5]. Des latences trop élevées peuvent même mener à l'échec d'un projet [6].

Les profileurs permettent de diagnostiquer les problèmes de performance les plus simples.

Toutefois, nous avons vu à la section 1.1.2 que, pour les problèmes qui résultent de l'interaction entre plusieurs composants ou qui se produisent de façon sporadique, l'utilisation d'un traceur est de mise. L'analyse de traces d'exécution est toutefois coûteuse en raison du grand volume d'information qu'elles contiennent. Une grande proportion de cette information n'est d'ailleurs souvent pas reliée au problème étudié. Les visualiseurs de traces facilitent la navigation à travers un grand nombre d'événements. Il requièrent toutefois une bonne connaissance du système analysé pour pouvoir juger si les résultats obtenus correspondent aux attentes. Il faut aussi avoir une idée préalable de la position du problème dans la trace. Il existe des systèmes experts capables de détecter automatiquement des problèmes. Cependant, la détection de problèmes spécifiques à un composant est possible uniquement si des règles ont été écrites pour celui-ci. Écrire et maintenir ces règles est coûteux. Il y a aussi des chances que des composants soient oubliés, menant à des diagnostics incomplets.

Plusieurs problèmes de performance se manifestent sous la forme de variations entre les durées de plusieurs exécutions d'une même tâche. Ces variations peuvent survenir dans plusieurs situations :

— **Mise à jour** d'une application, de librairies dynamiques ou du système d'exploitation.
  Exemple : Un serveur MySQL est migré d'une machine SmartOS vers une machine Linux. La performance des requêtes à la base de données est 30% inférieure sur la nouvelle machine [7].

— **Interaction** sporadique entre plusieurs tâches partageant une même ressource.
  Exemple : Deux tâches répétitives sont confinées au même processeur. La première s'exécute à 1 Hz et la seconde à 50 Hz. Occasionnellement, les deux tâches tentent de s'exécuter simultanément, ce qui augmente leurs durées.

— **Erreur** de programmation.
  Exemple : Des éléments sont ajoutés régulièrement dans une cache qui n'est jamais nettoyée. La durée des exécutions augmente graduellement, car elles doivent parcourir la cache linéairement [8].

— **Charge du système** différente.
  Exemple : Les écritures à une base de données sont protégées par un verrou. La durée des requêtes à un site Web augmente lorsqu'il y a un grand nombre d'utilisateurs concurrents en raison de la contention sur le verrou d'écriture.

Ce travail de recherche fait l'hypothèse que les traces d'une exécution longue et d'une exécution normale comportent des différences qui sont suffisantes pour expliquer la cause de la variation de performance. Un outil capable d'extraire automatiquement ces différences permettrait donc d'accélérer significativement le diagnostic d'une grande variété de problèmes

de performance.

Une trace de système parallèle contient normalement des événements appartenant à plusieurs exécutions distinctes. Pour faire une comparaison pertinente entre des exécutions, il faut démultiplexer ces événements. Notons aussi que nous souhaitons inclure dans notre analyse tous les facteurs pouvant influencer la durée totale des exécutions : nous devons donc être en mesure de suivre les dépendances entre fils d'exécutions répartis sur une ou plusieurs machines ainsi que les interactions avec le système d'exploitation.

Des exécutions distinctes peuvent comporter des différences sans que cela soit anormal. Pour minimiser la quantité de faux positifs générés par notre outil, nous souhaitons comparer des groupes d'exécutions plutôt que des exécutions individuelles. Cela permettra à notre outil de calculer la variabilité acceptable au sein d'un groupe d'exécutions normales.

1.3 Objectifs de recherche

Les travaux présentés dans ce mémoire visent à répondre à la question de recherche suivante :

> Est-il possible de concevoir un algorithme qui accélère le diagnostic de variations de performance en identifiant automatiquement les différences entre deux groupes de traces d'exécution ?

Pour répondre à cette question, nous tenterons d'atteindre les objectifs spécifiques suivants :

1. Développer une technique efficace pour collecter de l'information faisant le lien entre les événements de bas niveau du système d'exploitation et le code des applications en espace utilisateur.
2. Développer une méthode pour démultiplexer les événements d'une trace appartenant à des exécutions différentes.
3. Concevoir un algorithme capable d'extraire les différences significatives entre deux groupes d'exécutions.
4. Concevoir une interface permettant de spécifier des groupes d'exécutions à comparer et de visualiser leurs principales différences.
5. Valider à l'aide de traces d'exécution démontrant des variations de performance réelles que la méthode développée permet d'identifier rapidement les causes de ces variations.

1.4 Plan du mémoire

Au chapitre 2, on présente des travaux existants dans les domaines du traçage et de la comparaison de données de performance. Ensuite, le chapitre 3 décrit la méthodologie qui nous a permis de concevoir et évaluer la solution dont traite ce mémoire. Le chapitre 4 contient l'article « Diagnosing Performance Variations by Comparing Multi-Level Execution Traces » soumis à la revue IEEE Transactions on Parallel and Distributed Systems. Cet article détaille la solution que nous avons développée pour identifier automatiquement les causes de variations de performance entre plusieurs exécutions d'une même tâche. Il explique également comment cette solution nous a permis de corriger de réels problèmes de performance dans des logiciels libres et d'entreprises. Le chapitre 5 présente des résultats complémentaires et revient sur les critères d'évaluation définis lors de la méthodologie. Enfin, le chapitre 6 fait une synthèse des travaux et propose des améliorations futures.

CHAPITRE 2 REVUE DE LITTÉRATURE

Ce chapitre dresse un portrait de l'état de l'art en ce qui a trait à la collecte et à l'analyse de données sur la performance de systèmes parallèles complexes. Il décrit aussi des techniques permettant de comparer des ensembles de données volumineux. La solution proposée dans ce travail de recherche sera construite à partir des connaissances présentées dans ce chapitre.

2.1 Outils d'observation de systèmes parallèles

Le comportement d'un système parallèle est complexe à observer. En effet, la moindre perturbation introduite par un outil d'observation peut modifier complètement le comportement du système et rendre inintéressantes les données collectées. Cette section présente des outils permettant d'observer des systèmes parallèles avec un minimum de perturbations.

Dans le cadre de nos travaux, nous utiliserons ces outils pour collecter des données décrivant de manière détaillée plusieurs exécutions de tâches complexes. Nous nous baserons ensuite sur ces données pour trouver des différences entre les exécutions.

2.1.1 Traceurs

Un traceur est un outil capable d'enregistrer des événements survenant durant l'exécution d'un système. Les principaux traceurs peuvent enregistrer au sein d'une même trace des événements survenant dans le système d'exploitation (changement de contexte, écriture sur le disque, réception d'un paquet réseau...) et dans les applications en espace utilisateur (entrée dans une fonction, réception d'une requête HTTP...).

Linux Trace Toolkit Next Generation (LTTng)

LTTng [1] est un outil pour tracer les espaces noyau et utilisateur sur le système d'exploitation Linux. Son principal objectif est de minimiser les perturbations du système observé [9]. Chaque événement peut être activé individuellement. Une attention particulière a été apportée pour que l'impact sur la performance des points de trace désactivés soit imperceptible.

Lorsqu'un système est tracé avec LTTng, le système d'exploitation et les applications écrivent des événements dans des tampons en mémoire. Afin d'assurer une extensibilité à plusieurs cœurs, des tampons différents sont associés à chaque cœur. Cela élimine le besoin d'avoir

1. `https://lttng.org`

recours à des verrous. Lorsqu'un tampon est plein, un signal est envoyé à un démon. Le démon se charge de copier le tampon dans un fichier ou de l'envoyer sur le réseau.

LTTng propose aussi un mode *flight recorder* dans lequel les tampons ne sont pas consommés automatiquement. Les nouveaux événements écrasent plutôt continuellement les plus anciens. Une sauvegarde des tampons peut être déclenchée au besoin, par exemple lorsqu'une erreur survient ou lorsqu'une latence trop grande est détectée. Une trace des quelques secondes précédant l'incident devient disponible pour analyse. Les avantages du mode *flight recorder* sont de réduire les perturbations du système et d'éviter d'avoir à gérer le stockage de traces volumineuses. Un utilisateur peut écrire un programme personnalisé pour surveiller les conditions menant à la sauvegarde des tampons. Toutefois, pour détecter des latences au sein du système d'exploitation, il est plus simple et efficace d'utiliser le module `latency_tracker` [10]. Ce module propose une interface de programmation (API) pour signaler le début et la fin d'événements arbitraires. Au début d'un événement, une estampille de temps est sauvegardée dans une table de hachage *read-copy update* (RCU) [2]. Puis, à la fin de l'événement, sa durée totale est calculée. Si celle-ci dépasse un seuil prédéfini, une fonction de rappel est exécutée. Le surcoût de ce module sur l'application `hackbench` [3] est de 0.3% lorsque tous les événements sont générés par le même cœur et de 22% avec 32 cœurs.

Les événements du noyau capturés par LTTng sont générés par la macro `TRACE_EVENT` ou par le mécanisme d'instrumentation dynamique *kprobes*. La macro `TRACE_EVENT` est un moyen simple et universel d'ajouter de l'instrumentation statique dans le noyau Linux [11]. N'importe quel traceur peut s'y interfacer. Le mécanisme *kprobes* est plus flexible puisqu'il permet l'ajout d'instrumentation à des emplacements arbitraires sans recompilation. Toutefois, il est plus coûteux sur le plan de la performance.

Une application en espace utilisateur peut aussi générer des événements. Pour cela, un développeur doit insérer dans son code C ou C++ une macro propre à LTTng et lier son application avec la librairie `liblttng-ust`. Un événement est alors généré chaque fois que la macro est rencontrée. Une procédure similaire est offerte pour Java. Un avantage du traçage en espace utilisateur avec LTTng est qu'il ne requiert aucun appel système, ce qui lui permet d'être très performant [12]. Les événements sont écrits par les applications dans des tampons en mémoire partagée. Un démon unique est responsable de collecter les événements générés par toutes les applications instrumentées du système. Notons aussi que les estampilles de temps des événements en espace utilisateur sont cohérentes avec celles des événements du noyau.

2. `https://www.kernel.org/doc/Documentation/RCU/whatisRCU.txt`
3. `https://github.com/linux-test-project/ltp/blob/master/testcases/kernel/sched/cfs-scheduler/hackbench.c`

Il est possible d'automatiser l'instrumentation d'une application. Par exemple, on peut compiler des fichiers sources avec l'option `-finstrument-functions` [4] des compilateurs GCC et LLVM et lier l'application avec la librairie `liblttng-ust-cyg-profile` afin d'obtenir un événement à chaque entrée et sortie de fonction. Il a été démontré que le compilateur GCC subit un ralentissement par un facteur 3.22 lorsqu'il est ainsi instrumenté [9].

La fonctionnalité de « contexte » de LTTng permet d'ajouter dynamiquement des champs à des événements lors de la configuration d'une session de traçage. Ces champs peuvent fournir de l'information sur le processus ayant généré un événement (pid, priorité, nom) ou sur la valeur de divers compteurs de performance au moment où l'événement est survenu (nombre d'instructions ou de branchements exécutés, nombre de fautes de cache...). Des développements récents permettent aussi d'ajouter la pile d'appels d'une application en espace utilisateur en tant que contexte d'événements noyau [5]. Cela est fort utile pour comprendre dans quel contexte est fait un appel système.

Une limitation de LTTng est qu'il ne permet pas de faire efficacement le lien entre les événements noyau et le code des applications utilisateur. Pour faire ce lien, il faut soit ajouter de l'instrumentation manuelle (requiert du temps humain et risque de ne pas être exhaustif) ou utiliser l'option `-finstrument-functions` (requiert l'accès au code source et ralentit considérablement l'application).

Event Tracing for Windows (ETW)

ETW [6] est un traceur intégré au système d'exploitation Windows. Il permet le traçage des événements noyau fournis par Microsoft. Les développeurs peuvent aussi générer des événements à partir de leurs applications. Des applications populaires telles qu'Internet Explorer et Node.js [13] contiennent déjà de l'instrumentation ETW.

Un profileur est intégré à ETW [14]. Lorsque celui-ci est activé, des événements contenant la pile d'appels des applications en cours d'exécution sont générés périodiquement. Ces événements permettent par exemple de déterminer quelles fonctions s'exécutent dans un fil d'exécution au moment où un autre fil est en attente d'une ressource.

Les détails de l'implémentation de ETW ne sont pas tous connus. On sait cependant que les événements sont écrits dans des tampons circulaires en mémoire [15]. Les tampons peuvent être automatiquement copiés dans un fichier lorsqu'ils sont pleins ou la copie peut être faite sur demande (semblable au mode *flight recorder* de LTTng).

4. `http://gcc.gnu.org/onlinedocs/gcc-4.3.4/gcc/Code-Gen-Options.html`
5. `https://github.com/fdoray/lttng-modules/tree/addons-callstack`
6. `https://msdn.microsoft.com/library/bb968803.aspx`

Une limitation de ETW est qu'en raison de sa licence propriétaire, il n'est pas possible de créer de nouveaux événements noyau selon nos besoins.

DTrace

DTrace [7] est un traceur pour Solaris introduit en 2004 [16]. Depuis, il a été porté aux systèmes d'exploitation FreeBSD et Mac OS X [17]. Tout comme LTTng, il permet de tracer le noyau et l'espace utilisateur et vise un impact minimal sur la performance.

DTrace se distingue par sa capacité à associer des scripts en langage D aux points d'instrumentation. Cette fonctionnalité permet de spécifier dynamiquement des actions à effectuer lorsqu'un événement survient. Ces actions peuvent être utilisées pour enregistrer de l'information personnalisée dans la trace (par exemple, la pile d'exécution d'un processus), filtrer les événements devant être tracés (réduisant ainsi la taille de la trace) ou même faire des agrégations en ligne. Les scripts peuvent utiliser des variables pour conserver des données d'un événement à l'autre.

Une limitation de DTrace est qu'il est plus lent que LTTng : 6.42 fois pour le noyau [18] et 10 fois pour l'espace utilisateur [19].

2.1.2 Profileurs

Les profileurs produisent des statistiques décrivant l'exécution d'un système pour un intervalle de temps donné [20]. Par exemple, un profileur de processeur indique le temps de processeur consommé par instruction, fonction ou pile d'appels dans l'intervalle de temps analysé. Un profileur de mémoire indique la quantité de mémoire allouée par fil d'exécution, pile d'appels ou type d'objet.

Contrairement aux traceurs, les profileurs ne tiennent pas compte de l'ordonnancement des événements. Ainsi, avec un traceur, on pourrait déterminer dans quel ordre différents fils d'exécution ont acquis un verrou. Avec un profileur, on pourrait uniquement connaître le temps total passé en attente de verrou par chaque fil d'exécution.

Un avantage des profileurs par rapport aux traceurs est qu'ils produisent généralement moins de données. En effet, ils calculent des statistiques globales plutôt que de garder les détails de chaque événement. Cela leur permet de moins perturber le système analysé.

7. `http://dtrace.org/blogs/about/`

Perf

`perf`[8] a été conçu pour faciliter l'accès aux compteurs de performance du processeur [21]. Les compteurs de performance sont des registres spéciaux qui sont incrémentés lorsque des événements matériels surviennent (fautes de cache, erreurs de prédiction de branchement, cycles du processeur...). Dans son mode d'exécution le plus simple, `perf` produit un rapport indiquant le nombre de fois que divers compteurs ont été incrémentés dans un intervalle de temps donné (voir figure 2.1).

```
francois$ perf stat tar -xvf myfile.tar.gz

         84.523170 task-clock (msec)
               883 context-switches
                 7 cpu-migrations
               652 page-faults
       176,533,152 cycles
       103,589,310 stalled-cycles-frontend
       173,519,272 instructions
        29,784,016 branches
         1,185,831 branch-misses

       0.133966386 seconds time elapsed
```

Figure 2.1 Exécution de la commande `perf stat`

`perf` est aussi en mesure d'indiquer quelles instructions d'un programme ont provoqué les événements associés aux compteurs de performance. Étant donné que la plupart de ces événements surviennent à une fréquence très élevée, il serait impensable d'enregistrer l'instruction en cours d'exécution à chaque occurrence sans avoir un impact important sur la performance. Pour cette raison, `perf` demande une interruption pour une fraction des occurrences seulement. L'instruction ayant provoqué l'événement est capturée à l'intérieur de cette interruption. Statistiquement, chaque instruction devrait représenter une proportion similaire dans l'échantillon capturé et dans la population complète. Depuis sa création `perf` a été mis à jour pour supporter des événements logiciels (changements de contexte, fautes majeures, appels système...) en plus des événements du processeur. Il est même capable d'ajouter de l'instrumentation dynamique au noyau grâce à *kprobes*.

`perf` peut être configuré pour capturer périodiquement des piles d'appels pour l'ensemble d'un système (espaces utilisateur et noyau). Ce mode constitue un moyen efficace de repérer

8. `https://perf.wiki.kernel.org`

les fonctions qui consomment le plus de temps de processeur. Lorsque les *frame pointers* ne sont pas présents (par défaut depuis GCC 4.6.0 [9]), extraire les adresses de retour de la pile n'est pas trivial. `perf` enregistre donc 2 pages de mémoire dans la trace dans le but d'extraire les adresses de retour lors de la phase d'analyse [22]. Cela fait rapidement augmenter la taille de la trace.

Bien que `perf` soit principalement utilisé comme profileur, il peut associer des estampilles de temps aux événements qu'il enregistre et ainsi faire office de traceur [23].

Perf n'est pas conçu pour surveiller des systèmes en production [24].

`oprofile` [10] offre des fonctionnalités similaires à `perf`. Il affecte toutefois davantage la performance du système [25].

Google Performance Tools

Le profileur de processeur de *Google Performance Tools* [11] permet d'enregistrer la pile d'appels d'un programme à intervalles réguliers [26]. Il prend la forme d'une librairie que l'on injecte dans le programme à analyser par la variable d'environnement `LD_PRELOAD` [12]. L'injection d'une librairie peut être problématique : cette action peut uniquement être faite au démarrage du programme et peut modifier le comportement de celui-ci. Contrairement à `perf`, ce profileur ne peut pas analyser l'ensemble du système.

La librairie utilise le minuteur `ITIMER_PROF` [13] pour exécuter à intervalles réguliers une routine qui capture la pile d'appels. Plusieurs mécanismes sont disponibles pour capturer cette pile. Le mécanisme le plus performant a recours à la fonction `backtrace()` [14] fournie par Linux. Cette fonction produit toutefois des résultats incorrects lorsque l'application omet le *frame pointer* (c'est le cas pour les programmes compilés avec les options par défaut de GCC). Pour cette raison, le profileur offre aussi un mécanisme qui a recours à la librairie `libunwind` [15]. Celle-ci produit des résultats corrects même en l'absence du *frame pointer*, au prix d'une exécution plus lente. Le profileur peut être démarré en ligne de commande ou à l'intérieur même du programme analysé *via* une API C, ce qui est pratique lorsque l'on connaît la région de code que l'on souhaite analyser. Le résultat du profilage du processeur peut être visualisé sous la forme d'un graphe d'appels où la taille des nœuds représente le temps passé dans

9. `https://gcc.gnu.org/onlinedocs/gcc-4.6.0/gcc/Optimize-Options.html`
10. `http://oprofile.sourceforge.net`
11. `http://google-perftools.googlecode.com/svn/trunk/doc/cpuprofile.html`
12. `http://man7.org/linux/man-pages/man8/ld.so.8.html`
13. `http://linux.die.net/man/2/setitimer`
14. `http://man7.org/linux/man-pages/man3/backtrace.3.html`
15. `http://www.nongnu.org/libunwind`

une fonction. Les arêtes sont annotées pour montrer quels sont les principaux appelants de chaque fonction.

Google Performance Tools fournit aussi un profileur de mémoire [27] [16]. Comme le profileur de processeur, ce profileur prend la forme d'une librairie à injecter dans un programme par la variable d'environnement `LD_PRELOAD`. La librairie instrumente les fonctions allouant de la mémoire. Lors de l'exécution du programme, la pile d'appels associée à chaque allocation est capturée. Un outil d'analyse utilise cette information pour générer un graphe d'appels où la taille des nœuds est proportionnelle à la quantité de mémoire allouée. L'outil peut aussi comparer des profils de mémoire capturés à différents moments dans le cycle de vie d'un programme, dans le but de détecter des allocations de longue durée. Les principales limitations de cet outil sont qu'il requiert d'avoir recours à `tcmalloc` [17] pour les allocations de mémoire et qu'il gère mal les librairies qui font leur propre gestion de la mémoire (ex. : STL).

Java VisualVM

Java VisualVM [18] peut profiler la consommation de temps processeur et les allocations de mémoire d'une application Java [28]. Ce profileur instrumente le programme de manière à ce qu'il génère un événement à chaque entrée et sortie de fonction. Cela a un coût plus élevé que d'interrompre le programme à intervalles réguliers. Cette solution a toutefois l'avantage de donner le nombre exact d'appels à chaque fonction.

2.2 Analyse de traces d'exécution

Les traces d'exécution contiennent un très grand volume d'information brute. Pour les utiliser dans le cadre d'une analyse de performance, de fiabilité ou de sécurité, il est essentiel d'avoir recours à des outils automatisés. Ces outils sont capables de générer une représentation à plus haut niveau de l'information contenue dans la trace. À l'aide de cette représentation, l'utilisateur peut naviguer efficacement dans une trace et acquérir une compréhension du comportement global du système. Au besoin, il peut accéder aux événements bruts correspondants à une région d'intérêt.

Dans le cadre de nos travaux, la représentation à haut niveau pourra être utilisée en entrée d'un algorithme de comparaison de traces. En effet, une grande quantité de bruit est présente dans les événements bruts d'une trace. Un algorithme de comparaison peut trouver plus

16. `http://gperftools.googlecode.com/svn/trunk/doc/heapprofile.html`
17. `http://goog-perftools.sourceforge.net/doc/tcmalloc.html`
18. `http://visualvm.java.net`

aisément des correspondances pertinentes entre plusieurs traces, si on lui fournit des séquences d'événements de haut niveau. La tâche de l'algorithme est aussi plus facile lorsqu'un sens est associé aux événements qu'il traite.

2.2.1 Visualisation de l'état d'un système

L'outil d'analyse de traces TraceCompass[19] propose plusieurs vues de type « diagramme de Gantt » (figure 2.2). Dans ces vues, l'axe horizontal comporte des rangées associées à des ressources du système (fils d'exécution, processeurs, vecteurs d'interruptions), tandis que l'axe vertical affiche l'état de ces ressources en fonction du temps. L'utilisateur peut changer la position et le niveau de zoom de l'axe horizontal de façon interactive. TraceCompass offre aussi des vues de type « graphique XY » pour présenter l'utilisation d'une ressource (CPU, mémoire) en fonction du temps. Toutes les vues sont synchronisées selon l'axe du temps. Une limitation de TraceCompass est que chaque analyse est confinée à sa propre vue : un effort mental est requis pour combiner les informations que chaque analyse offre à propos d'une même ressource.

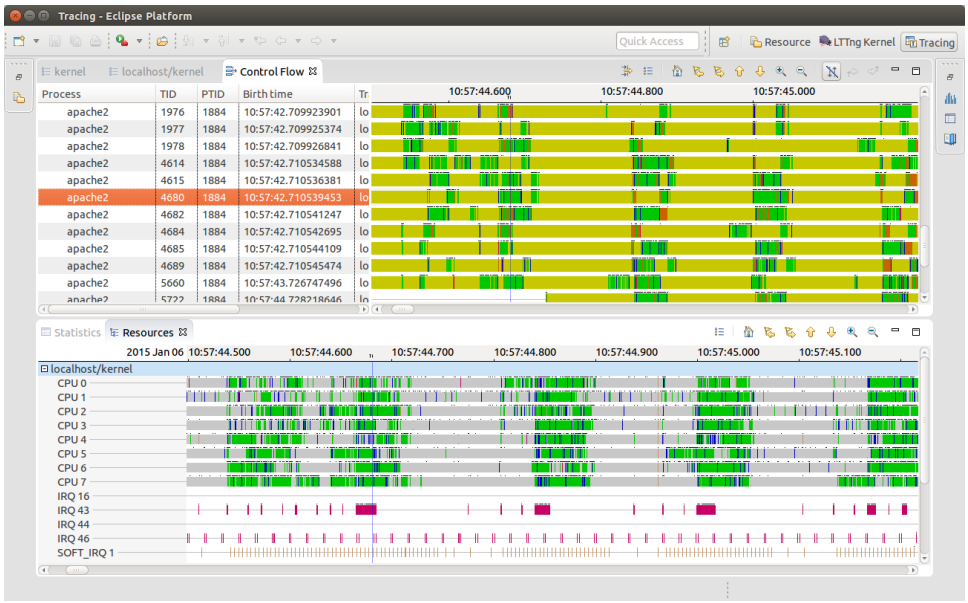


Figure 2.2 L'outil d'analyse de traces TraceCompass

19. `https://projects.eclipse.org/projects/tools.tracecompass`

Construction d'un historique d'états

Les historiques d'états associés aux vues de TraceCompass sont générés à partir des événements de bas niveau contenus dans des traces d'exécution [29]. Les traceurs fournissent généralement des événements spéciaux à partir desquels il est trivial de construire l'état initial d'un système (`lttng_statedump_*` pour LTTng et `DCStart` pour ETW). Une fois l'état initial du système connu, TraceCompass a recours à des machines à états pour générer un historique d'états. Les machines à états reçoivent séquentiellement les événements d'une ou plusieurs traces d'exécution. Lorsqu'un événement remplit les conditions associées à une transition accessible de l'état courant d'une machine, un changement d'état est généré et écrit dans un historique d'états. La figure 2.3 illustre la machine à états permettant de construire l'historique d'états d'un fil d'exécution à partir d'événements du noyau Linux. En plus de suivre l'état de ressources de bas niveau, il est possible de concevoir des machines à états pour suivre l'état d'un système complet. Par exemple, on peut détecter l'état « Soumis à une attaque de déni de service » [30].

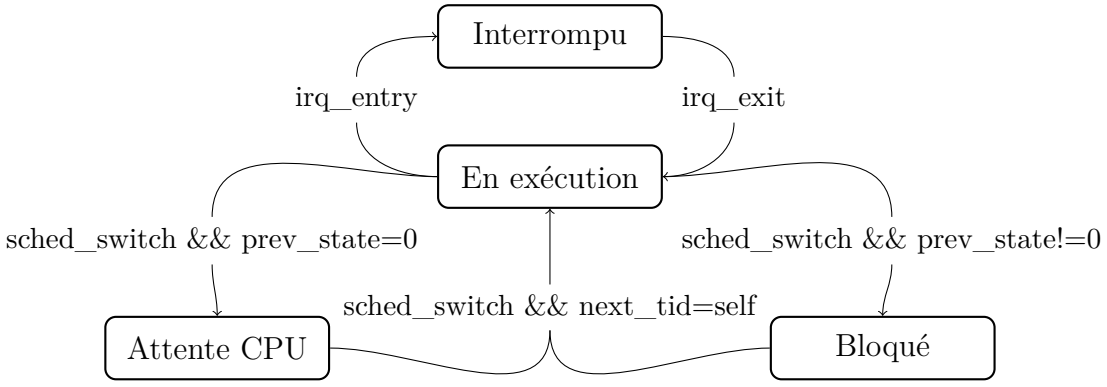


Figure 2.3 Machine à états pour un fil d'exécution

Spécification d'un modèle d'états

Un langage déclaratif a été développé pour décrire les machines à états de TraceCompass de manière flexible [31]. Ce langage permet la création de nouvelles analyses sans modifier le code de l'outil d'analyse. Il favorise aussi le découplage entre le modèle et les vues. Le langage a été utilisé avec succès pour créer de nouvelles vues de types « diagramme de Gantt » et « graphique XY » [32]. Les auteurs concèdent toutefois que le langage Java demeure plus puissant pour des analyses avancées (synchronisation de trace, analyse de chemin critique).

Modèle d'états multiniveaux

Un modèle d'états multiniveaux a été proposé pour faciliter l'exploration de traces volumineuses [33]. Dans ce modèle, un historique d'états à très haut niveau est présenté à l'utilisateur au niveau de zoom le plus reculé. Au fur et à mesure que l'utilisateur fait un zoom avant sur la trace, des états plus détaillés sont dévoilés. Ainsi, l'état « lecture des fichiers de configuration » se transforme progressivement en lectures de fichiers individuels puis en accès au disque. Les services de cartographie tels que Google Maps ont recours à des techniques similaires. Malheureusement, une analyse multiniveaux est plus complexe à concevoir qu'une analyse traditionnelle. Cette fonctionnalité risque donc de ne pas être utilisée pour les analyses *ad-hoc* et d'être réservée aux analyses les plus populaires.

Stockage d'un historique d'états

En raison de leur fine granularité, les historiques d'états présentés dans les vues de TraceCompass peuvent facilement nécessiter plusieurs gigaoctets de données pour quelques minutes de trace. Une telle quantité d'information ne peut pas être stockée en mémoire vive. TraceCompass utilise donc une structure de données nommée *state history tree* [34] qui est optimisée pour le stockage sur disque dur de valeurs associées à des intervalles de temps. Un disque dur est davantage optimisé pour les opérations faites séquentiellement que pour les opérations faites dans un ordre aléatoire. Pour cette raison, le *state history tree* n'a pas recours à des opérations de rebalancement comme celles que l'on retrouve dans les arbres AVL ou dans les arbres rouge et noir. Une fois qu'un noeud est écrit dans le *state history tree*, il n'est jamais modifié. Deux contraintes sont imposées aux insertions d'intervalles :

1. Les intervalles doivent être insérés en ordre croissant de bornes supérieures.
2. Les intervalles doivent avoir une faible longueur par rapport à l'étendue de tous les intervalles l'arbre. Ils doivent aussi être répartis uniformément dans le temps.

Chaque noeud de l'arbre peut contenir un certain nombre d'intervalles. Aussi, chaque noeud a une borne supérieure et une borne inférieure. Le noeud initial a une borne inférieure égale à zéro et une borne supérieure indéfinie. Des intervalles y sont insérés jusqu'à ce qu'il soit plein (figure 2.4).

Lorsqu'un noeud feuille est plein, sa borne supérieure est mise à jour et un nouveau noeud feuille est créé. La borne inférieure du nouveau noeud feuille est égale à la borne supérieure du noeud plein incrémentée de 1. Au besoin, on crée aussi de nouveaux noeuds parents pour conserver la structure d'un arbre binaire (figure 2.5).

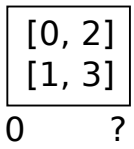


Figure 2.4 Noeud initial d'un *state history tree*

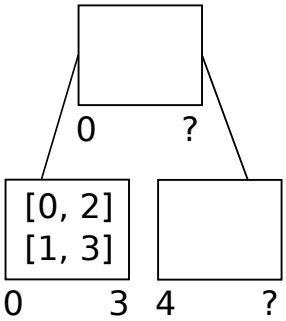


Figure 2.5 Gestion d'un noeud feuille plein dans un *state history tree*

Les insertions subséquentes se font dans le noeud le plus profond qui n'est pas plein et dont les bornes comprennent l'intervalle à insérer (figure 2.6).

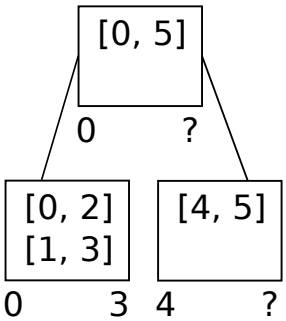


Figure 2.6 Insertion d'intervalles dans un *state history tree*

Lorsque le noeud racine est plein, tous les noeuds sont marqués comme « pleins ». Une nouvelle racine est créée et une branche est construite entre cette racine et un nouveau noeud feuille. La borne inférieure des noeuds de la nouvelle branche est égale à la borne supérieure du noeud plein incrémentée de 1 (figure 2.7). Ce cas devrait se produire rarement lorsque les contraintes présentées plus haut sont respectées.

Les auteurs démontrent que lorsque les contraintes présentées plus haut sont respectées, l'opération de recherche de tous les intervalles croisant un temps donné se fait en temps logarithmique. Ces contraintes sont respectées par la plupart des traces d'exécution. Des stratégies pour gérer les cas pathologiques ont été développées par [31].

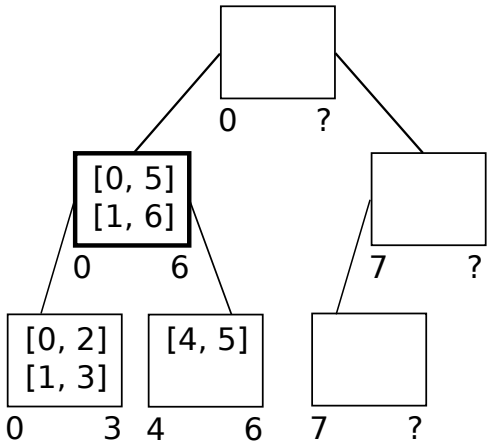


Figure 2.7 Gestion d'un noeud racine plein dans un *state history tree*

Pour plusieurs analyses, la taille du *state history tree* s'approche de celle de la trace source. Dupliquer les données ainsi peut être problématique. Dans un tel cas, on peut avoir recours à un *historique partiel*, c'est-à-dire un *state history tree* qui contient uniquement les intervalles croisant des points de sauvegarde. Les intervalles manquants peuvent être régénérés sur demande. Les requêtes dans l'historique partiel sont plus lentes que dans un historique complet d'un facteur constant proportionnel au nombre d'événements entre les points de sauvegarde.

2.2.2 Suivi de l'exécution d'une tâche

L'exécution d'une tâche dans un système informatique nécessite souvent la collaboration entre plusieurs processus d'une ou plusieurs machines. Par exemple, une recherche Google fait intervenir plusieurs milliers de machines [2]. Si l'on souhaite comprendre pourquoi l'exécution d'une tâche de ce type requiert plus de temps que souhaité, il faut être en mesure d'attribuer les événements, de traces d'exécution capturées sur différentes machines, à des exécutions de tâches spécifiques. Avec cette association, il devient possible de mesurer la latence et la consommation de ressources de chaque segment d'une exécution de tâche donnée.

Dans le contexte de nos travaux, on souhaite comparer la performance de plusieurs exécutions d'une même tâche. Pour ce faire, nous aurons besoin d'associer des caractéristiques à chaque exécution. Cela sera rendu possible par l'utilisation des techniques décrites dans cette sous-section.

Dapper

Chez Google, un identifiant unique est associé à chaque requête entrante. Cet identifiant est propagé dans tous les processus qui collaborent pour satisfaire une requête. Le traceur Dapper peut alors inclure cet identifiant dans les événements des traces qu'il enregistre localement sur chaque machine [2]. Pour réduire le surcoût du traçage, un nombre limité de requêtes sont tracées. Des démons rapatrient les traces enregistrées sur chaque machine et les insèrent dans une *BigTable*[20]. À partir de là, il est aisé de retrouver tous les événements appartenant à une requête donnée pour constituer un graphe annoté représentant l'ensemble de son exécution. L'interface graphique de Dapper offre la possibilité de faire des recherches parmi les traces capturées dans les dernières semaines. Des tableaux et histogrammes dressent un portrait de la performance des exécutions pour chaque type de requête capturé. Il est aussi possible d'inspecter une exécution individuelle en profondeur grâce à une vue de type « ligne du temps ». Dapper aide les ingénieurs de Google à diagnostiquer des problèmes et à trouver des opportunités d'optimisation. Notons que ce traceur enregistre des événements de haut niveau générés en espace utilisateur et non pas des événements du système d'exploitation. Aussi, il nécessite la propagation d'un identifiant de requête partout dans le système, ce qui n'est pas nécessairement possible lorsque des composants hétérogènes sont utilisés.

Zipkin[21] est une solution de traçage inspirée de la littérature sur Dapper [35]. La solution est développée par Twitter et distribuée sous licence libre.

Magpie

Magpie est un outil qui permet de suivre des requêtes dans un système distribué à partir de traces ETW [36]. Contrairement à Dapper, Magpie n'impose pas la propagation d'un identifiant unique dans tout le système. L'outil identifie plutôt les dépendances entre fils d'exécution en mettant en correspondance des paires d'événements de l'espace utilisateur portant un même identifiant. Par exemple, un événement d'envoi de appel de procédure à distance (RPC) est mis en correspondance avec l'événement de réception portant le même identifiant. La transitivité est appliquée sur les dépendances afin de retrouver tous les segments de fils d'exécution dont dépend une requête donnée. Magpie interprète ensuite des événements du système d'exploitation survenus sur ces segments de fils d'exécution afin d'attribuer la consommation de ressources du système (processeur, disque, mémoire) à des requêtes spécifiques.

En combinant l'information de plusieurs requêtes, Magpie peut construire une machine à

20. `http://dl.acm.org/citation.cfm?id=1365816`
21. `https://github.com/twitter/zipkin`

états probabiliste capturant le comportement habituel d'un système [37]. Lorsqu'une requête incorrecte est identifiée manuellement, la source du problème peut souvent être trouvée en inspectant les transitions de faible probabilité parcourues par la requête à l'intérieur de la machine à états.

Magpie établit un lien de dépendance entre des fils d'exécution uniquement lorsque des paires d'événements portant un même identifiant sont émises à partir de l'espace utilisateur. L'outil ne tire pas profit des événements du système d'exploitation pour déceler des dépendances inattendues, telles que de la contention pour l'utilisation du processeur, du disque ou d'un verrou. L'outil ne peut pas non plus repérer les dépendances créées au sein de librairies qui ne sont pas instrumentées. Enfin, Magpie n'offre aucun moyen de connaître les piles d'appels des fils d'exécution dont dépend une requête.

Chemin critique dans TraceCompass

TraceCompass propose une analyse de chemin critique permettant d'identifier précisément où est passé le temps durant l'exécution d'une tâche distribuée [38]. L'analyse permet de voir dans quelles machines et quels processus est passé le temps d'exécution, mais aussi où sont les délais liés à l'attente du réseau, du disque ou de verrous. Contrairement aux approches précédentes, cette analyse se base uniquement sur des événements du système d'exploitation Linux et ne nécessite pas l'ajout d'instrumentation spéciale dans les applications analysées. Cela assure son universalité.

Une difficulté rencontrée lors de la conception de cette analyse est l'interférence due aux verrous. Sur la figure 2.8, une arête horizontale pleine correspond à l'exécution d'un fil sur le processeur, une arête horizontale pointillée correspond à un fil bloqué et une arête verticale correspond à une dépendance entre fils. Un algorithme typique de calcul du chemin critique procéderait en calculant le plus long chemin entre A et E, en supposant un coût nul pour les arêtes correspondant à des blocages. Pour l'exemple de la figure 2.8, le chemin critique calculé serait A-B-G-H-I-D-E. Malheureusement, si la dépendance B-G est causée par la prise d'un verrou par le fil 2, rien ne nous dit que le segment G-H, n'aurait pas pu être exécuté avant A si le hasard avait permis au fil 2 d'acquérir le verrou avant le fil 1. Les auteurs de l'analyse de chemin critique de TraceCompass ont noté que ce non déterminisme ajoutait souvent du bruit aux résultats calculés. Ils ont donc développé une heuristique dans laquelle le chemin critique peut passer d'une tâche parente à une tâche enfant uniquement lorsque la tâche parente est bloquée. Le chemin critique calculé pour la figure 2.8 à l'aide de cette heuristique serait A-B-C-H-I-D-E.

L'intégration de l'analyse dans TraceCompass est un avantage, puisque cela permet de cliquer

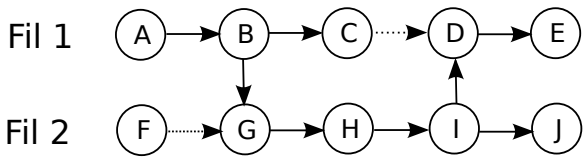


Figure 2.8 Graphe de dépendances entre fils d'exécution pour le calcul du chemin critique

sur un segment du chemin critique pour obtenir une vue détaillée de ce qui se passait sur l'ensemble du système à cet instant. Cela permet aussi de combiner des traces enregistrées sur plusieurs machines physiques ou virtuelles pour calculer un chemin critique distribué. L'analyse présente toutefois uniquement le point de vue du système d'exploitation : il n'est pas possible de connaître les fonctions utilisateur exécutées sur chaque segment du chemin critique. Aussi, aucune solution n'est proposée pour rapatrier automatiquement toutes les traces nécessaires au calcul d'un chemin critique distribué. Enfin, l'analyse requiert l'enregistrement de traces volumineuses. Il serait donc intéressant de pouvoir utiliser le mode *flight recorder* de LTTng pour faire une sauvegarde des tampons de traçage distribués lorsqu'une longue requête survient.

2.3 Mise en correspondance d'événements

La section précédente a présenté des solutions pour extraire d'une ou plusieurs traces tous les événements se rapportant à l'exécution d'une tâche. Cette section se concentre sur les techniques permettant de trouver les événements correspondants entre plusieurs exécutions d'une même tâche. Une fois que des événements correspondants sont trouvés, il devient possible de comparer leurs propriétés. On pourra par exemple se rendre compte qu'une fonction prend plus de temps, génère plus de fautes de cache ou lit plus de données sur le disque dans une exécution que dans une autre.

Dans un premier temps, on présente des méthodes pour mettre en correspondance des événements sans tenir compte de leur ordonnancement. Puis, on présente des techniques pour aligner des séquences ordonnées. Cela peut être utile lorsque les attentes ne sont pas les mêmes pour différentes occurrences d'un même type d'événement. Par exemple, on peut s'attendre à ce qu'une exécution contienne une courte lecture sur le disque suivie d'une lecture plus longue.

2.3.1 Mise en correspondance de profils d'exécution

[39] démontre que la mise en correspondance des métriques de plusieurs profils d'exécution

aide à repérer des fonctions qui s'adaptent moins bien à l'augmentation de la taille de l'entrée d'un programme. Pour cela, il collecte d'abord deux profils d'exécution d'un programme en faisant varier la taille de l'entrée. Les profils d'exécution donnent le temps passé dans chaque pile d'appels pour chaque exécution. L'auteur met ensuite en correspondance les temps d'exécution qui sont associés à des piles identiques dans les deux profils. Puis, il normalise les durées en les divisant par la taille de l'entrée du programme. Enfin, il obtient le ratio de changement de chaque métrique en divisant la valeur normalisée du second profil par celle du premier. Les métriques ayant les ratios de changement les plus élevés sont celles qui ont le plus de difficulté à s'adapter à l'augmentation de la taille de l'entrée du programme (complexité asymptotique supérieure à $O(n)$). L'auteur utilise une procédure similaire pour repérer les métriques qui s'adaptent moins bien à l'augmentation du nombre de processeurs utilisés par le programme. Cette procédure est uniquement applicable aux algorithmes pour lesquels l'utilisation du processeur est le facteur limitant. Il faut s'assurer au préalable que la variance des temps d'exécution des fonctions impliquées n'est pas trop élevée.

Les auteurs de [40] proposent également de mettre en correspondance les valeurs de différents profils qui sont associées à des piles identiques. Ils utilisent cette mise en correspondance pour générer un graphe d'appels dans lequel la taille des nœuds est proportionnelle à la différence de temps d'exécution d'une fonction entre deux profils. La solution n'est pas applicable telle quelle pour surveiller un système en production, puisque l'on peut observer des variations qui sont simplement dues à des charges différentes. Aussi, les profils d'exécution utilisés contiennent uniquement des temps totaux par pile d'appels. Si un long délai affecte un nombre limité d'utilisateurs, il est possible qu'il ne soit pas suffisant pour affecter significativement les totaux et ainsi ressortir dans la comparaison.

Les travaux de [41] décrivent un système complet utilisant la mise en correspondance de profils d'exécution pour diagnostiquer des régressions de performance. Leur système exécute des tests de performance sur chaque nouvelle version d'une application. Lorsqu'une régression est détectée, un profil d'exécution est capturé pour la version courante et pour la version précédant la régression. Les fonctions prenant plus de temps à s'exécuter dans le nouveau profil sont affichées à l'utilisateur. Les profils sont pris dans un environnement contrôlé, ce qui élimine le besoin de normaliser les données. Les résultats peuvent être peu pertinents si des fonctions ont été renommées ou si différents choix d'*inlining* ont été faits par le compilateur.

L'outil SPECTRAPERF développé par [42] utilise la mise en correspondance de profils d'exécution pour détecter les nouvelles opérations d'entrée et sorties effectuées par une application suite à un changement dans le code. Les auteurs collectent d'abord un profil de l'exécution d'une suite de tests d'une application. Le profil indique la quantité de données écrites et lues

sur le disque pour chaque appel de fonction. Afin de capturer les variations normales, les mêmes opérations sont répétées plusieurs fois par la suite de tests. À partir de ce profil, de limites minimales et maximales pour la quantité d'entrées et sorties de chaque fonction sont établies. Lorsque des changements sont introduits dans le code, un nouveau profil d'exécution est capturé en exécutant la même suite de tests. Les fonctions ne respectant pas les limites fixées par le profil de référence sont automatiquement détectées. La technique de *spectrum-based fault localization* (SFL) est aussi proposée pour évaluer la probabilité que des valeurs à l'extérieur des seuils soient réellement dues aux changements dans le code. Cet outil est limité à un type de problème très particulier.

Les travaux de [43] mettent en correspondance des profils d'exécution indiquant la fréquence à laquelle les branchements sont exécutés. Les auteurs adaptent des concepts de la théorie de l'information pour déterminer si les profils montrent un changement de comportement dans une application. Les traces de branchement ont un surcoût trop élevé pour être utilisées dans une analyse de performance.

2.3.2 Alignement de séquences d'ADN

La bio-informatique possède plusieurs méthodes pour repérer les ressemblances et les différences entre des séquences d'ADN. Nous donnons un aperçu de ces techniques, qui pourraient être adaptées à notre problème.

L'algorithme de Needleman-Wunsch utilise la programmation dynamique pour trouver l'ensemble minimal de substitutions, insertions et suppressions permettant de transformer une séquence ordonnée entre une autre [44]. On peut utiliser cet ensemble pour déduire quels sont les éléments qui n'ont pas changé entre les deux séquences. Pour exécuter l'algorithme, on commence par construire une matrice où les rangées sont associées à des éléments de la séquence A et les colonnes à des éléments de la séquence B. On remplit ensuite la matrice dans le sens de lecture à l'aide de la formule :

$$C(i,j) = min \begin{cases} C(i-1,j-1) \text{ si } A_i = B_i,\ C(i-1,j-1) + s \text{ sinon} \\ C(i,j-1) + d \\ C(i-1,j) + i \end{cases} \tag{2.1}$$

où s est le coût d'une substitution, d le coût de suppression d'un élément de la séquence A et i le coût d'une insertion. On suppose que la valeur des cases au-dessus de la matrice est égale à l'index de la colonne tandis que la valeur des cases à gauche de la matrice est égale à l'index de la rangée. Pour chaque case, on note quelle case a été utilisée dans le calcul de sa valeur

(dépendance de donnée). Une fois la matrice remplie, les paires d'éléments correspondants sont retrouvées en sélectionnant les cases associées à des éléments identiques le long d'un chemin suivant les dépendances de données de la dernière à la première case (voir figure 2.9). Si l'on souhaite aligner plus de deux séquences, on utilise plutôt l'algorithme Clustal W [45]. Le temps d'exécution polynomial de ces algorithmes peut être prohibitif pour aligner de longues séquences.

	A	C	G	T	A
A	↖ 0	← 1	← 2	← 3	← 4
G	↑ 1	↖ 1	↖ 1	← 2	← 3
G	↑ 2	↑ 2	↖ 1	← 2	← 3
T	↑ 3	↑ 3	↑ 2	↖ 1	← 2

Figure 2.9 Alignement de séquences avec l'algorithme de Needleman-Wunsch
Les dépendances de données sont illustrées par des flèches.

Les *dot plots* sont aussi des matrices où les rangées et les colonnes sont associées à des séquences d'ADN [46]. Un point noir est dessiné dans les cases correspondant à des éléments identiques. Les sous-séquences correspondantes se révèlent alors sous la forme de lignes diagonales solides. Cet outil aide à repérer des correspondances même lorsqu'il y a des permutations. Aussi, il met en évidence les sous-séquences répétées plusieurs fois dans les séquences étudiées. Toutefois, il sert uniquement à aider un humain à observer des tendances. Il est peu utile comme entrée d'un programme informatique.

2.3.3 Alignement de traces d'appels de fonctions

Une trace d'appels de fonctions est une trace contenant un événement pour chaque entrée et sortie d'une fonction. L'algorithme de programmation dynamique présenté à la section 2.3.2 a été réutilisé maintes fois pour l'alignement de traces d'appels de fonctions. Cela pose toutefois plusieurs problèmes. D'abord, l'algorithme ne tient pas compte de la structure hiérarchique des appels de fonctions. Il est illogique d'associer deux appels ayant le même parent dans une trace à deux appels ayant des parents différents dans une autre trace. Également, le temps d'exécution quadratique de l'algorithme le rend inutilisable pour des traces comportant des millions d'appels. En fait, dans le domaine de la biologie, des techniques alternatives ont été développées pour aligner des séquences d'ADN de grande taille [47].

Des travaux proposent d'adapter l'algorithme de programmation dynamique à la structure hiérarchique des appels de fonctions [48]. Dans ces travaux, l'algorithme est d'abord appliqué en considérant seulement les appels au premier niveau de la pile d'appels. Puis, il est appliqué

à nouveau de manière récursive à l'intérieur de chaque paire d'appels identifiée à l'étape précédente. Cette approche produit des résultats plus respectueux de la structure hiérarchique des appels de fonctions, tout en réduisant le temps total d'exécution de l'algorithme. Cependant, elle demeure vulnérable aux boucles qui peuvent causer un très grand nombre d'appels sous un même parent. Aussi, deux versions d'un programme peuvent avoir des différences qui empêchent l'exécution de cet algorithme : modifications dans l'ordre d'appels de fonctions, fonctions renommées, fonctions subdivisées, ajout d'un niveau dans la pile d'appels... Enfin, ce type d'alignement a un intérêt limité pour analyser des programmes parallèles.

Un article de [49] décrit une méthode pour filtrer les événements de traces d'appels de fonctions avant d'effectuer un alignement. La méthode combine d'abord tous les appels consécutifs à une même fonction (éliminant ainsi les boucles). Puis, tous les appels à des fonctions utilitaires (identifiées par leur nom commençant par « set » ou « get ») sont retirés. Des traces de plus petite taille sont ainsi obtenues. Les auteurs proposent ensuite d'extraire des traces filtrées de courtes sous-séquences ayant une fréquence d'occurrence élevée. Enfin, ils comparent les métriques de sous-séquences à fréquence élevée retrouvées dans différentes traces. Les auteurs ont choisi de comparer uniquement les métriques de sous-séquences parfaitement identiques. Par exemple, les métriques associées à l'exécution de la sous-séquence A, B, C ne sont pas comparées au métriques associées à l'exécution de la sous-séquence A, B, Z, C. Aussi, les auteurs font une comparaison entre des sous-séquences sorties de leur contexte. Ils ne permettent pas à l'utilisateur de connaître les appels faits avant ou après ces sous-séquences. On pourrait très bien imaginer que les appels faits avant une sous-séquence donnent des indications sur le volume de données à traiter par les appels de la sous-séquence, et expliquent ainsi les différences de performance mises en évidence par la comparaison.

PerfDiff propose d'ignorer l'ordonnancement des appels et de construire des *calling context trees* (CCTs) [50]. Les nœuds de deux CCTs sont mis en correspondance à l'aide d'un algorithme de programmation dynamique qui supporte l'ajout et le retrait de fonctions au milieu des piles d'appels. Puisque tous les appels partageant une pile identique sont combinés, le temps d'exécution de l'algorithme est raisonnable. Des pointeurs vers les événements de la trace sont conservés à l'intérieur des nœuds des CCTs. Cela permet d'extraire des métriques pour chaque appel de fonction (durée de l'appel, nombre de fautes de cache...) et de comparer les distributions de valeurs entre plusieurs exécutions. L'outil requiert une trace d'appels de fonctions qui est coûteuse à produire en termes de performance.

IntroPerf s'attaque au problème de générer des traces d'appels de fonction avec un surcoût minimal et sans modifier les programmes observés [51]. Les auteurs tirent profit du fait que plusieurs traceurs peuvent capturer la pile d'appels d'un programme lors des appels systèmes

et qu'un grand nombre de problèmes de performance sont liés aux interactions avec le système d'exploitation. Leur outil déduit les temps d'entrée et de sortie de chaque fonction à partir d'une trace d'appels système avec piles d'appels. Pour cela, les auteurs supposent que l'entrée dans une fonction survient la première fois qu'elle apparaît dans la pile d'appels d'un appel système. Puis, la sortie survient lorsque la fonction cesse d'être présente dans la pile d'appels des appels système. Les fonctions ne faisant pas d'appels système ne sont pas visibles par cet outil. Les longs délais sans appels système provoquent des résultats invalides, ce qui peut se révéler frustrant.

2.3.4 Visualisation de l'alignement entre des traces d'appels de fonctions

Un algorithme d'alignement de traces d'appels de fonctions produit une liste de paires d'appels correspondants. Pour que cela soit utile, il faut offrir à l'utilisateur une manière efficace de visualiser les correspondances même lorsque le nombre d'événements est très grand.

TraceDiff est un outil destiné à aider les utilisateurs à repérer et comprendre les différences entre deux traces d'exécution [52]. L'utilisateur choisit d'abord manuellement des segments à analyser dans deux traces. Puis, une vue de comparaison affiche les appels de chaque segment de trace sous forme de profils d'horizon (*icicle plots*) (voir figure 2.10). Des tubes connectent les appels correspondants entre les traces. Les tubes sont groupés à l'aide d'une technique inspirée du *hierarchical edge bundling* dans le but d'augmenter la lisibilité de la vue. Les couleurs sont utilisées pour mettre en évidence les permutations.

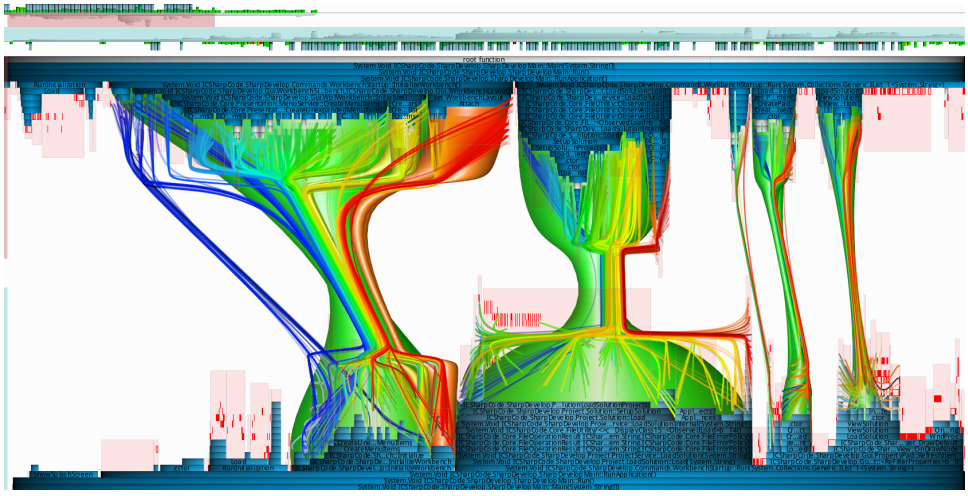


Figure 2.10 La comparaison de traces avec TraceDiff
Image tirée de [52], reproduite avec permission.

TraceDiff affiche des tubes pour chaque paire de fonctions correspondantes, ce qui peut représenter beaucoup de bruit lorsque l'on cherche simplement les différences. Il n'est pas

clair si la vue demeure fluide lorsque des traces volumineuses sont comparées. Aussi, TraceDiff ne peut pas comparer plus de deux exécutions simultanément. Les auteurs de Magpie (voir section 2.2.2) surmontent cette limitation en superposant les machines à états probabilistes générées par leur outil. Ainsi, ils peuvent comparer deux groupes comprenant chacun un nombre arbitraire d'exécutions.

Paraver est un autre outil permettant de visualiser l'alignement entre des traces d'appels de fonctions [53]. Il affiche les traces sous forme de lignes du temps colorées selon la fonction appelée. Lorsque plusieurs traces sont ouvertes simultanément, les auteurs prétendent que les couleurs permettent à l'oeil de repérer les fonctions correspondantes. Cette technique est bien entendu inutilisable pour des traces comportant un grand nombre d'appels.

2.3.5 Alignement de traces d'instructions

Une trace d'instructions est plus détaillée qu'une trace d'appels de fonctions, puisqu'elle permet de suivre chaque branchement fait dans le code d'un programme. Des travaux proposent une méthode pour trouver les instructions correspondantes entre des traces d'instructions de différentes versions d'un même programme [54]. Les traces utilisées contiennent les instructions exécutées, les adresses référencées et les valeurs produites. À partir de cette information, un graphe dans lequel les nœuds correspondent à des instructions et les arêtes à des dépendances de données est construit. Les nœuds de deux graphes sont mis en correspondance par un algorithme utilisant comme critères de similarité le type et l'ordonnancement des instructions ainsi que les dépendances de données. L'algorithme tient compte du fait qu'une même opération peut être effectuée par un nombre différent d'instructions dans les différentes versions du programme. La solution est appliquée pour trouver des erreurs introduites par un optimiseur et pour repérer des clones logiciels.

L'alignement de traces d'instructions est utilisé par [55] dans le domaine de la sécurité informatique. Dans un premier temps, les auteurs mettent à profit l'alignement de traces pour déterminer comment l'exploitation de failles de sécurité affecte le comportement de divers programmes. Puis, ils utilisent l'alignement pour identifier les conditions qu'un logiciel malveillant vérifie avant de s'activer (par exemple, la présence d'un antivirus).

La capture de traces d'instructions est très coûteuse. À moins d'avoir un moyen pour filtrer les données enregistrées, cette technique n'est pas adaptée pour analyser la performance d'un système. En effet, l'action de capturer la trace affecterait trop les résultats obtenus.

2.3.6 Alignement de traces de systèmes parallèles

Dans un système parallèle, il n'y a pas nécessairement de lien causal entre les événements survenant dans le même intervalle de temps sur différents fils d'exécution. Cela complique le calcul d'un alignement entre deux traces d'un tel système.

[56] ont développé une méthode pour trouver des correspondances entre des traces de systèmes parallèles dans le but de détecter des attaques. Les auteurs de cette méthode font l'hypothèse qu'il est difficile pour un attaquant de concevoir une attaque fonctionnant simultanément contre plusieurs machines ayant des configurations variées. En effet, les failles exploitables sur chaque configuration sont différentes. Pour cette raison, ils proposent d'exécuter une application sur plusieurs machines hétérogènes soumises au traçage. Selon leur hypothèse, les différentes machines devraient réagir différemment en cas d'attaque. Ils peuvent donc détecter des attaques contre un système distribué en calculant un score de distance entre des traces prises sur des machines ayant des configurations différentes. Pour pallier les différences inhérentes à l'utilisation de configurations différentes, ils proposent de transformer les événements de bas niveau en abstractions de plus haut niveau. Par exemple, les appels système servant à ouvrir un fichier et y écrire des données deviennent simplement une « écriture », un concept qui existe sur tous les systèmes d'exploitation. Aussi, pour contrer les variations d'ordonnancement des événements dues au parallélisme, les auteurs choisissent de ne pas tenter un alignement complet des traces. Ils extraient plutôt de chaque trace des n-grammes (séquences d'événements répétées plusieurs fois dans une trace). Ils calculent ensuite un score de distance entre deux traces à partir des différences entre leurs ensembles de n-grammes. Cette méthode est vulnérable aux attaques de type *mimicry*, c'est-à-dire que rien n'empêche un attaquant d'inclure dans son attaque des appels système dont le seul but est d'imiter la distribution de n-grammes du système original [57, 58]. Aussi, les auteurs se contentent de calculer un score de distance entre deux traces. Ils ne cherchent pas des correspondances ou divergences pertinentes à présenter à un utilisateur.

Les travaux de [59] consistent à comparer des traces dans le but de détecter des patrons de communication inattendus dans des systèmes embarqués. Ils expliquent qu'un système embarqué peut être modélisé par une machine à états dont les transitions sont annotées avec des communications attendues. Des réseaux de neurones ou des chaînes de Markov sont en mesure de retrouver une telle machine à états à partir d'un volume suffisant de traces jugées normales. Toutefois, en présence de parallélisme, la machine à états est plus complexe que nécessaire. Les chercheurs proposent donc de démultiplexer les tâches représentées dans les traces étudiées afin de générer une machine à états distincte par tâche. Pour cela, ils calculent le domaine de fréquence associé à chaque type d'événement à l'aide d'une transformée de Fou-

rier. Puis, ils forment des grappes d'événements partageant des composantes fréquentielles rapprochées. Des tests démontrent que cette méthodologie permet de générer des automates couvrant 69% des événements générés par le groupe motopropulseur d'une voiture. Les automates ne contiennent aucune information sur les délais des opérations. Aussi, ils servent à étudier les communications entre composants et non pas le comportement interne de ces composants. Enfin, la méthode produit de nombreux faux positifs.

Groupement d'exécutions d'un système parallèle

Spectroscope est un outil qui génèrent des graphes pour représenter les communications survenues au cours de l'exécution de tâches [60]. L'outil regroupe les graphes similaires en grappes grâce à l'algorithme des k-moyennes [22]. Les caractéristiques utilisées par l'algorithme sont les suites d'événements et la latence associées à chaque arête des graphes de requêtes. Les grappes correspondent ainsi à différents types de requêtes où à différentes manières de traiter un même type de requête. Un utilisateur peut avoir une vue d'ensemble du comportement d'un système en visualisant un représentant de chaque grappe.

Les auteurs de Spectroscope prétendent que les problèmes intermittents sont masqués lorsque des compteurs de performance globaux sont utilisés pour surveiller un système. En formant des grappes, ces problèmes se révèlent sous la forme de grappes comportant peu de requêtes.

La cause d'un problème peut souvent être identifiée en comparant des grappes similaires de manière à repérer leurs différences. Pour faciliter cette comparaison, les auteurs de Spectroscope représentent les composantes principales des requêtes de chaque grappe au moyen d'un arbre. Les arbres des grappes d'intérêt sont ensuite comparés visuellement.

Il est aussi intéressant d'exécuter Spectroscope sur des traces capturées à différents moments et de comparer la répartition des requêtes dans les grappes entre ces différents moments. On peut par exemple observer que davantage de requêtes sont placées dans la grappe correspondant à un accès en cache, au détriment d'une grappe correspondant à une requête à un serveur distant.

Spectroscope utilise uniquement des événements de haut niveau générés par l'espace utilisateur pour construire ses graphes (pas d'événements noyau). Il est donc inefficace pour des applications qui n'ont pas été instrumentées. Il est aussi limité quant au nombre de requêtes pouvant être classifiées.

22. `http://fr.wikipedia.org/wiki/Algorithme_des_k-moyennes`

2.4 Métriques pour analyser la performance

Une grande variété de métriques peuvent être utilisées pour évaluer la performance d'un système. Par exemple, les travaux de [61] décrivent comment des métriques liées à l'utilisation du processeur, de la mémoire, du système de fichiers ou du réseau peuvent être utilisées pour diagnostiquer des problèmes de performance. Toutes ces métriques peuvent être extraites de traces du noyau Linux capturées par LTTng. Le livre de [62] détaille une méthode complète pour détecter et diagnostiquer des problèmes à partir de métriques. Il explique comment faire le lien entre l'utilisation et la saturation de diverses ressources et la manifestation de problèmes courants.

Une façon intéressante d'utiliser les métriques de performance est de comparer les valeurs obtenues dans différents contextes. On peut par exemple comparer les valeurs obtenues avant et après un changement dans une application. On peut aussi comparer les valeurs obtenues sur des machines ayant des configurations différentes ou à différents moments de la journée. Enfin, il est pertinent de comparer les valeurs obtenues à des valeurs attendues.

Cette section présente des techniques pour comparer des métriques de performance.

2.4.1 Spécification d'attentes pour des métriques

Un expert peut définir des attentes quant aux valeurs des métriques de performance d'un système donnée. Ces attentes sont parfois inscrites dans un contrat (on les appelle alors *service-level objectives*). Une fois que l'on sait que les métriques d'un système répondent aux attentes, il demeure pertinent de les surveiller. Si elles passent en dessous des seuils établis, on voudra sans doute trouver les différences entre une exécution où le système respecte les seuils et une exécution où il ne les respecte pas.

Spécification d'attentes pour les métriques de tests de performance

Les développeurs du navigateur Chromium ont à leur disposition un tableau de bord de suivi de la performance [23]. Celui-ci les aide à s'assurer que la performance des multiples fonctionnalités du logiciel ne régresse pas au fil des versions. Les données présentées par ce tableau de bord sont issues de l'exécution continuelle de tests de performance par un ensemble de machines ayant des configurations variées. Chaque test de performance est écrit manuellement par un développeur. Un test spécifie des actions à effectuer, des métriques à collecter, une révision de référence et des seuils acceptables pour la variation des métriques.

23. `https://chromeperf.appspot.com`

Des machines de tests récoltent continuellement les métriques spécifiées dans les tests pour la révision de référence et pour la dernière révision disponible dans l'outil de gestion de configuration. Lorsqu'un écart dépassant le seuil spécifié est obtenu entre les deux versions, une alerte est affichée dans le tableau de bord. Le tableau de bord permet le suivi des actions effectuées pour répondre à une alerte. La spécification des attentes par rapport à une révision de référence, plutôt qu'en valeurs absolues, permet d'éviter les fausses alertes lors de changements dans l'environnement de tests. Les métriques collectées par cet outil sont généralement de très haut niveau. Elles sont suffisantes pour détecter une régression, mais pas pour la diagnostiquer. Pour effectuer le diagnostic, les développeurs sont contraints de reproduire le problème localement. Cela est problématique pour les cas qui dépendent de la configuration de la machine de tests ou qui surviennent sporadiquement.

Spécification d'attentes pour les métriques associées à des opérations

Pip est un outil permettant aux programmeurs de spécifier dans un langage déclaratif leurs attentes quant aux performances de certaines opérations exécutées par un logiciel [63]. Les spécifications sont traduites en machines à états non déterministes qui sont capables de reconnaître des opérations complexes et de vérifier que les métriques collectées durant ces opérations répondent aux attentes. Pip est utilisé avec succès pour vérifier des contraintes liées à la synchronisation entre fils d'exécution et à l'échange de messages entre composants. Une limitation de cette solution est que les développeurs n'évaluent pas toujours adéquatement le temps requis pour effectuer une opération donnée. Aussi, l'écriture manuelle de contraintes par les développeurs est une tâche ardue qui risque de mener à une couverture incomplète du système.

2.4.2 Maîtrise statistique des procédés

Les cartes de contrôle de [64] servent à s'assurer qu'un processus est en contrôle. Elles ont été inventées pour minimiser les variations dans la production d'équipements de téléphonie. Sur une carte de contrôle, on retrouve un seuil minimal et maximal pour la valeur d'une métrique. Ces seuils sont calculés à partir de l'historique des valeurs de la métrique. Lorsque des valeurs se retrouvent à l'extérieur de ces seuils, on conclut qu'un facteur empêche le processus d'être en contrôle.

Un article de [65] souligne que ces cartes ne peuvent pas être utilisées telles quelles pour contrôler la performance de systèmes informatiques. En effet, les métriques étudiées dans ce domaine ne suivent généralement pas une distribution unimodale normale. Aussi, une variation des métriques proportionnelle au taux de requêtes n'est pas anormale. Les auteurs

proposent deux transformations pour contourner ces limitations. D'abord, ils appliquent un filtre pour éliminer les modes non significatifs et respecter l'hypothèse de normalité. Ensuite, ils appliquent une transformation linéaire pour absorber les variations associées aux changements du taux de requêtes. Cette stratégie leur permet de repérer des défauts introduits manuellement dans divers systèmes distribués. L'élimination artificielle des valeurs aberrantes signifie que les problèmes sporadiques ne seront pas détectés.

[41] ont aussi recours à des cartes de contrôle pour repérer les variations de performance dans des logiciels. Cependant, ils les utilisent pour surveiller des métriques collectées lors de l'exécution de tests contrôlés, ce qui élimine le besoin d'appliquer des transformations aux valeurs étudiées. Les auteurs remarquent que les cartes de contrôle de Shewhart repèrent difficilement les changements subtils dans la moyenne d'une métrique. Pour cette raison, ils les combinent avec des cartes de contrôles à somme cumulative (CUSUM). Évaluer la performance uniquement dans un environnement contrôlé n'est pas suffisant, puisqu'il a été démontré que plusieurs problèmes se manifestent uniquement dans un environnement de production [1].

2.4.3 Comparaison de distributions de valeurs

Plusieurs tests statistiques permettent de vérifier s'il est plausible que deux échantillons soient tirés de la même distribution. Pour l'analyse de performance de systèmes informatiques, ces tests permettent de déterminer s'il existe une différence statistiquement significative entre la distribution des valeurs d'une métrique dans un contexte de référence et dans un contexte d'intérêt. De tels tests donnent de meilleurs résultats que de simples comparaisons de moyennes lorsque les distributions comparées comportent plusieurs modes ou sont asymétriques.

Test de χ^2 à deux échantillons

Un test de χ^2 à deux échantillons permet de vérifier l'hypothèse que deux échantillons de valeurs catégorisées sont tirés d'une même distribution. Ce test est utilisé par [66] pour identifier le composant fautif lorsque la performance d'un service se dégrade. Les auteurs proposent de surveiller un système distribué en permanence à l'aide d'un nombre limité de métriques. Lorsque les valeurs de métriques provenant d'un nœud du système s'éloignent de seuils prédéfinis, un traceur est activé dynamiquement sur le nœud fautif et sur un nœud sain. Ensuite, des tests de χ^2 à deux échantillons sont appliqués sur les métriques supplémentaires collectées par le traceur afin d'identifier celles qui sont le plus susceptibles d'expliquer le problème observé. Puisqu'un répartiteur de charge devrait envoyer un nombre de requêtes similaire à tous les nœuds du système, les variations dues à des différences de charge sont éliminées.

Aussi, l'activation dynamique du traçage évite de perturber le système en permanence. Cette stratégie repose sur l'hypothèse que le traceur peut collecter des informations pertinentes en étant activé seulement après la première détection du problème. Cette hypothèse peut être vérifiée pour certains problèmes liés à la saturation des ressources, mais est moins applicable pour les problèmes sporadiques liés à des erreurs de programmation. La stratégie est aussi inefficace pour détecter des erreurs de configuration lorsque toutes les machines sont identiques.

Traces de fréquences

Les distributions de métriques de latence sont souvent complexes et ne peuvent pas être bien décrites par une moyenne et un écart-type. Elles peuvent comporter plusieurs modes (par exemple, un mode associé à une lecture en cache et un mode associé à une lecture sur disque) et des valeurs aberrantes (par exemple, dues à un lien réseau déconnecté). Toutes ces caractéristiques sont bien visibles sur une courbe de distribution. Les traces de fréquences [67] sont une manière de présenter simultanément plusieurs courbes de distribution d'une métrique (voir figure 2.1). Cet affichage simultané facilite la comparaison visuelle des distributions. Les traces de fréquence ont été utilisées avec succès pour décrire de manière exhaustive l'impact de changements à la configuration d'un système. Notons toutefois que les traces de fréquence présentent une seule métrique à la fois. Pour les utiliser, il faut avoir une idée préalable des métriques pertinentes à analyser.

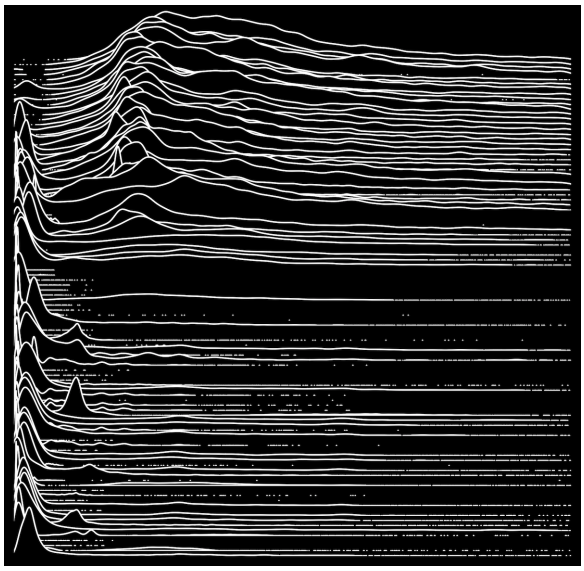


Figure 2.11 Traces de fréquences de la latence de lectures sur disque sur différents serveurs
Image tirée de [67], reproduite avec permission

2.4.4 Construction de modèles de relations entre métriques

Les travaux de [68] consistent à construire un modèle autorégressif avec entrées exogènes (ARX) décrivant le comportement attendu des métriques d'un système. Ce modèle n'est pas basé sur les valeurs absolues des métriques, mais plutôt sur les relations qui existent entre elles. À partir d'un historique de valeurs de métriques, des invariants portant sur les relations entre métriques sont générés. Les auteurs révèlent que ce type de relation est très présent dans les systèmes distribués, souvent même entre composants distincts. Par exemple, si le traitement d'une requête par un serveur applicatif nécessite en moyenne trois requêtes à la base de données, on aura la relation « le nombre de requêtes à la base de données est le triple du nombre de requêtes au serveur applicatif ». Semblablement, pour un service de lecture de vidéos en continu (*streaming*) qui écrit régulièrement des métriques dans une base de données au cours de la lecture des vidéos, on aura une relation du type « il y a 1 écriture dans la base de données par tranche de 1 Mo de données transférées sur le réseau ». Les auteurs démontrent qu'une fois le modèle généré, des problèmes peuvent être détectés en vérifiant le respect des invariants. Le modèle sert aussi à faire de la planification de la capacité. D'autres travaux révèlent que si les conditions influençant les relations entre métriques (par exemple, la disponibilité de valeurs en cache) ne sont pas prises en compte, les modèles risquent d'être imprécis [69]. Aussi, les modèles doivent être mis à jour régulièrement, ce qui requiert beaucoup de ressources de calcul [70].

Chez Netflix, des métriques du système et des applications sont agrégées dans le tableau de bord Atlas qui permet d'observer leur évolution. L'outil Mogul effectue des analyses statistiques sur ces mêmes métriques afin de repérer des corrélations fortes [71]. Les modèles de corrélation générés sont utilisés par les ingénieurs pour mieux comprendre l'impact de leurs applications sur l'utilisation de ressources. Ils révèlent aussi le couplage qui existe entre des applications. Tous les composants de Netflix doivent être en mesure de transmettre leurs métriques à Atlas via un mécanisme homogène.

2.4.5 Détection d'anomalies dans les métriques en présence de tendances saisonnières

Dans un système distribué, les métriques suivent souvent des tendances annuelles, hebdomadaires ou quotidiennes. Cela complexifie la détection de valeurs anormales. Chez Twitter, une technique nommée *E-Divisive with Medians* (EDM) a été développée pour détecter un passage d'un état stable à un autre dans les valeurs d'une métrique, malgré la présence de variations saisonnières [72]. Les changements d'état détectés par cette méthode peuvent être causés par le déploiement continu de code, des tests A/B ou encore des pannes de longue

durée. Il est aussi intéressant de repérer les valeurs anormales qui ne sont pas associées à un changement d'état permanent. Cela peut être causé par des erreurs logicielles intermittentes, des attaques informatiques ou encore des événements extérieurs qui modifient le taux d'utilisation d'un service. Twitter a publié un algorithme faisant ce type de détection en se basant sur l'algorithme généralisé *Extreme Studentized Deviate* (ESD) [73]. Ces outils n'identifient pas la cause des variations. Il est désagréable de faire ce travail manuellement pour finalement se rendre compte que l'outil a rapporté un faux positif. Pour minimiser le taux de faux positifs, Skyline [24] combine les forces de plusieurs algorithmes de détection [74].

2.5 Conclusion de la revue de littérature

Les traceurs ont recours à des technologies avancées pour enregistrer les événements survenant à plusieurs niveaux dans un système informatique avec un surcoût minimal. Des structures de données spécialisées permettent une navigation fluide dans les traces volumineuses qu'ils génèrent. Aussi, plusieurs stratégies permettent d'extraire d'une ou plusieurs traces tous les événements se rattachant à une exécution de tâche donnée. Il n'existe toutefois pas de solution automatisée pour analyser les variations entre les événements survenus lors de différentes exécutions d'une même tâche. Ce travail de recherche s'appuie sur les différentes méthodes de comparaison de données de performance présentées dans cette revue de littérature pour combler cette lacune. Au chapitre suivant, nous présentons la méthodologie suivie pour concevoir notre solution.

24. `https://github.com/etsy/skyline`

CHAPITRE 3 MÉTHODOLOGIE

La conception d'un outil de résolution de problèmes de performance doit être encadrée par une méthodologie rigoureuse pour s'assurer qu'il soit adapté aux besoins de l'industrie. L'outil doit en effet s'attaquer à des problèmes concrets auxquels font face les développeurs de logiciels de haute performance, tout en respectant les contraintes propres à ce domaine. Ce chapitre détaille la procédure suivie pour mener notre étude.

3.1 Définition du problème

Nous avons visité quatre entreprises développant des logiciels de haute performance et nous avons scruté les répertoires de bogues de plusieurs logiciels libres afin de bien cerner le contexte dans lequel notre solution pourrait être appliquée. Cette démarche nous a permis d'identifier des causes de variations de performance auxquelles notre outil devrait être en mesure de s'attaquer. Celles-ci ont déjà été introduites à la section 1.2. Nous présentons aussi ici des particularités du domaine qui devront guider nos choix de conception.

3.1.1 Les systèmes analysés sont méconnus des développeurs.

Les logiciels de haute performance ont fréquemment un code source énorme, qui n'est pas forcément compris dans son ensemble par chacun de ses développeurs. Il ne serait pas approprié de concevoir un outil d'analyse de performance nécessitant l'ajout manuel d'instrumentation dans les fonctions à surveiller. Les développeurs seraient à coup sûr rebutés par l'idée de devoir parcourir le code source à la recherche d'endroits stratégiques pour insérer des points de traces. Les plus téméraires risqueraient de ne pas penser à instrumenter certaines fonctions critiques, ce qui mènerait à une analyse incomplète. L'instrumentation de librairies tierces est quant à elle souvent impossible.

Les développeurs avec lesquels nous avons discuté préfèrent une approche dans laquelle un outil intelligent surveille l'ensemble d'un système et met lui-même en évidence le code fautif lorsqu'un problème est détecté.

Plusieurs outils existants se basent uniquement sur les événements provenant du noyau du système d'exploitation pour analyser la performance des applications. Ils ne nécessitent aucune instrumentation en espace utilisateur, ce qui permet leur universalité. Toutefois, cela peut rendre difficile l'identification du code source ayant provoqué les problèmes détectés.

3.1.2 Les systèmes analysés comportent des interactions entre de multiples composants.

Les développeurs possèdent des outils efficaces pour évaluer la performance de fonctions individuelles. Lorsqu'ils se tournent vers des outils plus avancés, c'est souvent parce qu'ils font face à des problèmes impliquant des interactions entre plusieurs fils d'exécution, avec le système d'exploitation ou avec le matériel. Il est donc crucial que nous soyons capables de bien représenter ces interactions multiniveaux dans notre outil d'analyse, même lorsque les développeurs ne soupçonnent pas leur existence.

À notre connaissance, il n'existe pas d'outil pouvant retrouver toutes les sources de latences dans un système parallèle et les associer au code source pertinent.

3.1.3 Les systèmes à analyser sont victimes de problèmes difficiles à reproduire.

Les développeurs rencontrent parfois des problèmes qui se manifestent rarement ou qui se reproduisent uniquement lorsque des conditions très particulières sont réunies. Grâce à leur faible surcoût, les traceurs peuvent être activés en tout temps sur de systèmes en production pour capturer de tels problèmes. On peut ainsi obtenir la séquence des événements s'étant produits autour du comportement problématique et l'analyser en détail dans un visualiseur de traces.

Toutefois, des développeurs nous ont confié qu'ils avaient trouvé fastidieux de chercher manuellement les différences entre une exécution fautive et de multiples exécutions correctes dans un visualiseur de traces. Ils ont exprimé le souhait d'avoir un outil effectuant cette comparaison automatiquement.

3.1.4 Les systèmes à analyser s'exécutent sur des plates-formes diversifiées.

Les développeurs d'applications de haute performance ont à travailler sur des systèmes d'exploitation et des architectures variés. Être forcé d'apprendre le fonctionnement d'outils d'analyse de performance différents pour chaque environnement est un inconvénient majeur. Ainsi, un outil pouvant présenter de façon similaire des données de performance capturées dans des environnements variés serait bien accueilli.

3.1.5 Les développeurs ont peu de temps à consacrer à l'analyse de performance.

Les entreprises valorisent grandement le développement de nouvelles fonctionnalités. La performance des applications est aussi un requis extrêmement important, mais les développeurs

sont souvent surchargés par leurs autres tâches et n'ont que très peu de temps à y consacrer. Tout outil qui limite le temps humain consacré à diagnostiquer des problèmes de performance est donc le bienvenu.

Un souhait formulé par plusieurs développeurs que nous avons rencontrés est d'avoir un outil évaluant automatiquement la performance de leur application de façon périodique tout au long du développement. Lors d'une régression de performance significative, l'outil produirait un rapport détaillant les changements survenus dans le comportement de l'application, ce qui permettrait de trouver la source du problème rapidement.

Un autre souhait qui nous a été transmis est d'avoir un outil d'analyse de la performance nécessitant un minimum de configuration. Les développeurs souhaitent qu'un outil puisse leur révéler pourquoi leur application n'a pas la performance attendue sans avoir à se demander quels types d'événements devraient être enregistrés.

3.2 Conception de la solution

Le laboratoire de recherche sur les systèmes répartis ouverts et très disponibles (DORSAL) de Polytechnique a développé de nombreuses technologies pour capturer et analyser des traces d'exécution de façon efficace. Des implémentations de ces technologies sont disponibles sous licence libre. Nous nous appuierons sur ces implémentations pour accélérer le prototypage des différents composants de notre contribution.

D'abord, nous utiliserons le traceur LTTng pour capturer des traces d'exécution. Ce traceur peut capturer avec un faible surcoût des événements issus du système d'exploitation et de l'espace utilisateur, ce qui convient à notre objectif de détecter des problèmes survenant à plusieurs niveaux et de les associer au code applicatif fautif. Notons toutefois que, pour obtenir des événements des applications en espace utilisateur, il faut qu'elles aient été instrumentées. Cela devra sans doute être amélioré.

Ensuite, nous utiliserons la librairie TigerBeetle [1] pour lire les traces au format Common Trace Format (CTF) produites par LTTng. Cette librairie permet de définir des fonctions à exécuter pour chaque type d'événement pouvant être lu à partir d'une trace. Tout au long de la lecture, on a accès à un arbre d'attributs représentant l'état courant du système. Aussi, la librairie propose un modèle de traitement de traces par lots. Ce modèle est bien adapté à notre désir d'accumuler des données issues de plusieurs traces enregistrées dans des contextes différents.

Nous réutiliserons l'algorithme de calcul du chemin critique proposé par [38] et intégré à

1. `https://github.com/eepp/tigerbeetle`

TraceCompass pour identifier tous les fils d'exécution affectant le temps de complétion d'une tâche. Il a été démontré que cet algorithme produisait des résultats exacts dans la plupart des cas sans nécessiter d'instrumentation en espace utilisateur.

Enfin, nous aurons recours à des librairies développées par d'autres groupes. Par exemple, nous utiliserons `libunwind` pour capturer la pile d'appels des applications en espace utilisateur. Nous utiliserons la librairie D3.js [2] pour connecter des données de performance à des vues. Nous utiliserons la librairie `crossfilter` [3] pour permettre aux utilisateurs d'appliquer des filtres sur ces mêmes données de façon efficace.

Le code de tous les outils que nous développerons sera publié sous une licence libre afin qu'il puisse être réutilisé par d'autres travaux de recherche dans le futur.

Au chapitre suivant, nous présentons l'article « Diagnosing Performance Variations by Comparing Multi-Level Execution Traces » qui détaille le fonctionnement de notre solution. L'article discute également de l'efficacité de notre solution, pour diagnostiquer de *réels* problèmes de performance, et de son surcoût.

2. `http://d3js.org`
3. `https://github.com/square/crossfilter`

CHAPITRE 4 ARTICLE 1: DIAGNOSING PERFORMANCE VARIATIONS BY COMPARING MULTI-LEVEL EXECUTION TRACES

Authors

François Doray
École Polytechnique de Montréal
francois.pierre-doray@polymtl.ca

Michel Dagenais
École Polytechnique de Montréal
michel.dagenais@polymtl.ca

Index terms – Performance analysis, Tracing, Software visualization, Concurrency, Operating systems.

Submitted to IEEE Transactions on Parallel and Distributed Systems.

4.1 Abstract

Tracing allows the analysis of task interactions with each other and with the operating system. Locating performance problems in a trace is not trivial because of their large size. Furthermore, deep knowledge of all components of the observed system is required to decide whether observed behavior is normal.

We introduce TraceCompare, a framework that automatically identifies differences between groups of executions of the same task at the userspace and kernel levels. Many performance problems manifest themselves as variations that are easily identified by our framework. Our comparison algorithm takes into account all threads that affect the completion time of analyzed executions. Differences are correlated with application code to facilitate the correction of identified problems. Performance characteristics of task executions are represented by a new data structure called enhanced calling context tree (ECCT).

We demonstrate the efficiency of our approach by presenting four case studies in which TraceCompare was used to uncover serious performance problems in enterprise and open

source applications, without any prior knowledge of their codebase. We also show that the overhead of our tracing solution is between 0.2% and 9% depending on the type of application.

4.2 Introduction

Performance is a critical requirement for many applications. Long delays are among the main sources of user frustration [4] and have a significant impact on revenue [75]. Despite that, it is often difficult for a single developer to understand all the factors that influence the performance of an application. This is mainly due to the multiple levels of abstraction that are supposed to ease software development: frameworks, operating systems and virtualization. Furthermore, diagnosing performance problems is hard because they don't necessarily trigger error conditions and often can't be easily reproduced.

Tracing is a technique that consists of recording events during the execution of a system. Events have a timestamp, a type and a payload. Tracing is well suited to analyze performance problems. Popular tracers achieve low overhead, which allows them to be enabled on production systems to capture bugs that occur infrequently. Tracing provides detailed chronological information, unlike profiling that only provides global statistics for a given time range. A trace can, for example, reveal what happened on the whole system during a long system call issued by a process of interest. However, unless userspace applications are carefully instrumented, it is hard to relate the low-level events of a trace to the higher-level logic. Also, it is hard to locate abnormal behavior in the overwhelming amount of information contained in a trace, without deep knowledge of the observed system.

Our objective was to build a tool that automatically highlights differences between two sets of executions of the same task. The comparison must take into account all factors that have an impact on performance, including off-CPU wait time and interferences between processes. Also, it must relate any difference found with the relevant source code, a requirement that is crucial to ease the diagnosis and correction of problems [76]. Our tool should allow a developer that doesn't have a full knowledge of a system to discover and fix performance variations caused by factors such as configuration changes, programming errors or sporadic competition between tasks.

This paper introduces TraceCompare, a trace comparison tool and underlying algorithms that achieve all these goals. After reviewing relevant existing work, we describe a data model to summarize performance characteristics of task executions. We then explain how to record detailed multi-level events during an execution and how to process them to generate a database of executions based on our data model. We introduce a GUI that facilitates the

identification of differences between groups of executions. We describe four *real* performance problems found in enterprise and open-source applications and explain how they were diagnosed using TraceCompare. Finally, we measure the overhead incurred by each component of our solution and show that it is low enough for use on production systems.

4.3 Related Work

4.3.1 Analyzing Execution Traces

Our group proposed using a state history to simplify and accelerate the analysis of large traces [34]. A state history is built incrementally while a trace is read. It keeps track of various attributes such as the current running thread on each CPU or the total number of bytes read from the disk since the beginning of the trace. An efficient data structure has been designed to allow fast queries about the value of an attribute at a given time within a state history that exceeds the size of the main memory. TraceCompass [1] relies on this data structure for its interactive views. State histories allow the computation of system metrics for any given time interval in logarithmic time [77].

4.3.2 Extracting Task Executions from a Trace

Our group developed a critical path algorithm to efficiently retrieve all execution segments that contribute to the latency of a task [38]. It uses the `sched_wakeup` event of the Linux kernel to identify dependencies between multiple threads on the same host (e.g. a thread waiting for another thread to release a mutex, write data to a pipe, send a signal, etc.). Also, it uses TCP packets matching to identify dependencies between threads that span multiple hosts. The analysis works well with any kind of application because it requires no userspace instrumentation. While this tool indicates in which threads time is spent, it does not relate this information to userspace functions, which makes it difficult to use for fixing application code. Also, the interactive view shows only one execution at a time, which is impractical to compare latency among a great number of executions. Perfscope provides a heuristic to find userspace functions associated with events from a kernel trace [1]. However, the results are not always accurate.

A portion of the requests received by Google are traced by Dapper [2]. The tool reconstructs the flow of individual requests using events generated by the company's communication libraries. Developers can access summary statistics or inspect individual requests through a Web interface. Filters help them find executions that contain abnormal latencies. Dapper

1. `http://projects.eclipse.org/projects/tools.tracecompass`

relies on the fact that all communications go through a Google's library and isn't designed for heterogeneous environments.

4.3.3 Comparing Task Executions

The "Frames" mode of Chrome developer tools presents the processing time of frames using bar charts [78]. Slow frames are shown as tall bars. Colors within each bar indicate in which states time was spent (JavaScript, rendering, GC...). Developers can determine why a frame took more time than others by comparing the distribution of time per state between different frames. The scope of this tool is limited to the browser. It doesn't take into account interactions between processes. A similar view was developed to compare the states in which real-time tasks spend their time from a kernel point of view [79].

A differential flame graph is a visualization tool to compare two CPU profiles [80]. It shows stack frames as superposed rectangles. The width of a rectangle is proportional to the time spent in a stack frame within the second profile. The color shows the difference between the time spent in a stack frame in the two profiles. Because a profile only provides total counts for a given period, this tool cannot reveal the cause of infrequent latencies. Flame graphs have been adapted to show different metrics, but they have never presented all the work done on the critical path of a task.

Comparisons of CPU profiles have been used to diagnose performance variations between multiple versions of the same application [41], evaluate the impact of configuration changes [81] and assess the scalability on an application with respect to the size of its input [39].

TraceDiff is a visualization tool to compare two function call traces [52]. A function call trace contains an event for every function entry and exit. TraceDiff displays two such traces simultaneously as mirrored icicle plots. Corresponding function calls are connected by hierarchical edge bundles. This facilitates the identification of individual function calls that didn't take the same time or weren't executed in the same order in the two traces. However, it doesn't allow simultaneous comparison of many executions to reveal trends. Also, generating function call traces is costly. Other researchers have developed a heuristic to get an approximate function call trace with low overhead [51]. Efficient algorithms to compute the optimal alignment between two function call traces have also been developed [48].

Comparison of system call traces recorded on different operating systems has been used to detect intrusions [56]. Since some variations are inherent to the use of different operating systems, the authors transform low-level events into higher-level concepts to allow a meaningful comparison. Their algorithm computes a correlation score for two traces, but doesn't

show individual differences. Also, it doesn't take timing into account.

4.3.4 Statistical CPU Profiling

A statistical CPU profiler is a tool that can record the full call stack of a program at regular intervals. This reveals which functions use more CPU time, with minimal overhead. While statistical profilers don't provide the chronological order of events, which is required to analyze interactions between multiple tasks, some of these techniques could be reused to get insight into the logic of userspace programs, without requiring manual instrumentation.

The CPU profiler of Google Perftools is a dynamic library, to link with programs to analyze [26]. In its initialization phase, it registers a `SIGPROF` signal handler and starts an `ITIMER_PROF` timer. The timer is decremented whenever a thread of the process consumes CPU time. When a thread reaches the expiration of the timer, the `SIGPROF` signal handler is invoked and captures the thread's call stack. Alternatively, `perf` leverages support from the Linux operating system to profile program [82].

Capturing the call stack of a program, when compiled with the frame pointer, is just a matter of following a linked list formed by base pointers pushed on the stack. However, in order to keep the `ebp` register available for computation, GCC does not preserve the frame pointer for x86-64 binaries, since version 4.6.0 [2]. The `.eh_frame` sections of these Executable and Linkable Format (ELF) binaries provide rules to restore the register values of the previous (oldest) stack frame from any instruction [83]. A call stack can be captured by applying these rules to retrieve the value of the instruction pointer at each stack frame. The `.eh_frame` section is present, even when debugging information is stripped, because it is required to handle exceptions.

The `libuwind` library implements the logic to capture call stacks using rules from the `.eh_frame` section. For good performance, it recognizes stack frames that use a standard layout, and caches that information to handle them with an optimized algorithm when they are encountered again [84]. Google Perftools relies on `libunwind` to capture call stacks online. Instead, `perf` records two pages of memory and runs `libunwind` offline [22]. This reduces the computation overhead, but leads to huge profile files.

A CPU profile can be represented using a calling context tree (CCT); a data structure introduced by [85] and reused by [50, 76]. In a CCT, each node represents a call stack. The children of a node associated with a call stack C of size n are associated with call stacks of size $n+1$ prefixed with C. The root of the CCT represents the empty call stack. Each node

2. `https://gcc.gnu.org/onlinedocs/gcc-4.6.0/gcc/Optimize-Options.html`

is annotated with the time spent in its associated call stack. Unlike a call tree which contains a distinct node for each individual function call, a CCT combines all calls associated with the same call stack into a single node. A CCT does not show interactions between different threads.

4.4 Solution

In this section, we present the design and implementation of TraceCompare. First, we describe a data model to store performance characteristics of task executions in a database. Second, we explain how to efficiently record the information required to build this database through tracing. Third, we explain algorithms used to build the database from traces. Finally, we present the user interface that highlights differences between groups of task executions. The general architecture of the tool is summarized in Fig. 4.1.

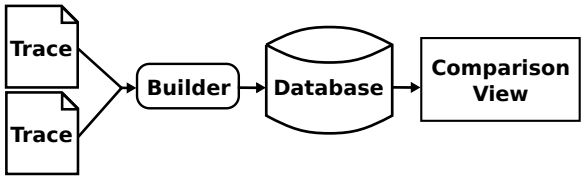


Figure 4.1 Architecture of TraceCompare

4.4.1 Data Model

We introduce a new data structure called enhanced calling context tree (ECCT) to represent the performance characteristics of task executions. An ECCT can describe any kind of latency that occurs during a task execution as well as the detailed context of each latency. Our database of executions and the algorithms that power our GUI rely on this data structure.

For executions that spend all their time on the CPU and involve a single thread, the structure of the ECCT is identical to that of a CCT. We annotate each node with the time spent in the associated call stack during the analyzed execution. Optionally, more annotations such as the number of page faults or amount of memory allocated in the call stack can be added.

For executions that contain off-CPU wait times or that involve multiple threads (possibly distributed on multiple hosts), the CCT is enhanced so that all factors that contribute to the total time of an execution are taken into account. First, for any off-CPU wait time, an artificial function named after the wait reason (timer, block device, network, preemption...) is pushed on the stack. Second, whenever a thread is waiting for another thread, the stack

of the second thread is concatenated to that of the first thread. This rule can be applied recursively as many time as required when there is a chain of dependencies between threads. The dependencies can be either direct (a thread is waiting for another thread to compute a result or perform an action) or indirect (a thread is waiting for another thread to release a resource such a the CPU or a mutex).

One might be concerned about the case where indirect dependencies between two different task executions lead to the concatenation of their ECCT. A safeguard could be added to stop the recursion in such a case. However, in our experience, it was rather useful to be able to see which parts of an execution could block other executions. Programs that use a limited number of worker threads have ECCTs of reasonable size, even when a lot of requests are queued.

Fig. 4.2 shows the expected ECCT for a sample sequence of events.

4.4.2 Tracing Task Executions

We now describe the techniques used to gather the information required to build the data model that we just described.

The LTTng [3] tracer can record events from the Linux kernel and from userspace applications into a single trace [9]. It is also designed to have a minimal overhead on traced systems. It is therefore well suited to our goal of collecting all the factors that contribute to the execution time of tasks in production environment.

Execution Delimiters Events

A single trace usually contains events belonging to multiple executions that occured in series or in parallel. To allow a comparison between different executions, it is necessary to demultiplex these events. To do so, the trace must provide a way to identify the start and end points of each execution. Sometimes, existing events of the Linux kernel can be used for this purpose. For example, the `syscall_exit_accept` event (generated when a connection is accepted on a socket) and the `syscall_entry_shutdown` event (generated when a connection is closed) correctly delimit requests received by an Apache server. An advantage of using existing events is that no access to the source code is required. When no existing events can act as delimiters, LTTng-UST probes must be inserted statically in the source code. Different probe types can be used to delimit different execution types. That way, our analysis tool is able to tag executions when they are inserted in the database so that the GUI only allows

3. `https://lttng.org`

Time	Thread 1	Thread 2
1	Call A	
2	Call B	
3	*Wait thread 2*	Call X
4		Return X
5		Call X
6		*Wait block device*
7		Return X
8	Return B	
9	Call X	
10	Return X	
11	Return A	

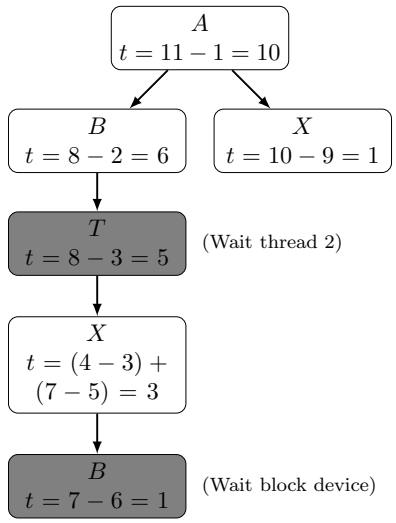


Figure 4.2 ECCT for a Sample Sequence of Events

comparison of executions of the same type.

Critical Path Events

When an execution involves a single thread running 100% of the time on a CPU, it is trivial to retrieve its events from the trace: we simply look for events between the start and end points on the execution's thread. However, in parallel systems, most executions require collaboration between multiple threads. TraceCompare relies on the critical path algorithm proposed by our group and introduced in section 4.3.2 to find all thread segments that contribute to the total time of an execution. This algorithm requires a few events of the Linux kernel to work properly.

The `sched_switch` event indicates that a new thread starts running on a CPU. Its payload gives the `tid` of the new thread and tells whether the previous thread was preempted or blocked. The `sched_wakeup` events is emitted when a blocked thread becomes ready to run. The reason why the thread was blocked can be deduced from the context in which the `sched_wakeup` event is emitted. For example, if the event is emitted from a block device related interrupt, it means that the waked up thread was waiting for a block device request. The `irq_handler_entry`, `irq_handler_exit`, `hrtimer_expire_entry`, `hrtimer_expire_exit`, `softirq_entry` and `softirq_exit` events allow to keep track of interrupts handled on each CPU in order to correctly interpret `sched_wakeup` events. The `inet_sock_local_in` and `inet_sock_local_out` events provide the sequence number and flags of TCP packets. Using these fields, incoming and outgoing packets can be matched to uncover dependencies between threads that communicate through TCP.

Call Stacks Events

TraceCompare must make it easy to relate performance variations, between multiple executions of the same task, to erroneous logic in userspace applications. LTTng allows developers to insert custom tracepoints in their code to correlate kernel events with the logic of userspace applications. This is, however, not suited to the requirements of TraceCompare. Indeed, we want to build a solution that developers can use on systems that they don't fully understand. Requiring to create tracepoints would represent a major maintenance burden and developers would inevitably forget to instrument important parts of their applications. Requiring instrumentation of third-party libraries would be even harder.

TraceCompare gets insight into userspace applications without any manual instrumentation by capturing their call stack in two different contexts. The first context in which the stack is captured is at the expiration of a timer that is decremented when a thread is running, a technique already used by statistical profilers. However, that doesn't provide information about call stacks that block the thread. An interesting observation is that most blockings occur within system calls. For example, a thread can block while it waits for a mutex to be available (`futex()`) or for data to be read from a file (`read()`). Therefore, also capturing the call stack on system calls gives a more complete portrait of the contexts in which a thread spends its time. In addition to block within system calls, a thread can block within page faults. Because the operations permitted within a page fault handler are limited (in particular, it is not safe to emit a signal), we decided not to capture the call stacks of page faults for the initial version of TraceCompare.

Statistical CPU profilers can keep track of the number of captured samples per call stack

using an hash table. This is sufficient to generate summary statistics for the full analyzed period. However, our goal is to provide per-execution statistics and we don't know beforehand to which execution a thread segment belongs to. Furthermore, during our offline analysis, we need to be able to correlate kernel events to userspace stacks and to combine multiple userspace stacks occuring simultaneously on different threads. For that reason, a timestamped LTTng-UST event is emitted every time a stack is captured. Stacks captured while userspace code is running produce a `cpu_stack` event while stacks captured during system calls produce a `syscall_stack` event.

To get the stacks of a process captured, the `lttng-profile` dynamic library must be preloaded into it. The library uses the same technique as Google Perftools to generate a `SIGPROF` signal at regular intervals, from which `cpu_stack` events are generated.

The `SIGPROF` signal must also be sent to capture the userspace stacks associated with system calls. Unfortunately, experimentations show that handling a signal can take longer than the duration of some fast system calls. Therefore, sending a signal and capturing the stack for each system call can add a prohibitive overhead to programs making a lot of short system calls. The problem is that, unlike statistical CPU profiling, which generates a fixed number of events per time unit and per thread, the overhead of this instrumentation is proportional to the number of system calls. To solve that, we decided to capture the userspace stack only for system calls with a duration above a threshold. This effectively sets a reasonable upper limit on the rate of events generated by each thread. Because performance is generally not affected by very short system calls, this doesn't undermine the accuracy of our analysis. We found that a 100 µs threshold was a good compromise between precision of the analysis and low overhead.

The duration of system calls is tracked by a kernel module. When it is loaded, the `lttng-profile` library registers the process with the kernel module through an `ioctl` interface. The kernel module uses the `TRACE_EVENT` mecanism [11] to register callbacks that are executed at the entry and exit of system calls. On system call entry, the kernel module checks whether the issuing process is registered. If so, the current timestamp is saved in a RCU hash table with the source thread id as the key. On system call exit, the duration of the system call is computed, if the start time is available in the hash table. If the duration is higher than a threshold, a `SIGPROF` signal is sent to the source thread. `lttng-profile` catches the signal and captures the call stack, as soon as the control goes back to userspace.

The first time that a thread does a system call, an entry is added to the hash table and it is reused until the thread exits. Therefore, most of the time, tracking the duration of a system call simply requires one update and one lookup in the hash table, two very efficient

operations.

Our `SIGPROF` signal handler uses `libunwind` to capture call stacks. This library is the state of the art to capture call stacks when the frame pointer is absent. We contributed two important optimizations to `libunwind` in order to accelerate the backtrace operation even more. First, we removed system calls that blocked signals while shared data structures were accessed. This was necessary to guarantee the reentrancy of `libunwind`. However, since we always invoke the library within `SIGPROF` signal handlers, and since Linux doesn't allow nested signals of the same type, it is valid to remove that protection in our case. Second, we replaced a loop that restored the value of all registers by more efficient code that restores only the registers that are really needed.

The call stack events generated by our library contain sequences of addresses, instead or human-readable function prototypes. This reduces the tracing overhead and the size of the trace. However, for analysis purposes, full symbols are required. Therefore, we record events that provide the base address of dynamic libraries. Those allow us to transform captured addresses into library offsets. Then, we parse ELF binaries offline, to retrieve function prototypes.

4.4.3 Building a Task Executions Database

We described how the data required by our analysis can be collected through tracing. We now explain how to process this data to build a database of task executions. The ECCT data structure introduced in section 4.4.1 is used to represent the performance characteristics of each task execution in the database.

The TraceCompare builder reads the events of a trace in chronological order. When it encounters an event associated with the beginning of an execution, the current timestamp and current thread are saved in a list of pending executions. Then, when the event marking the end of the execution is encountered, the following steps are performed:

1. Compute the critical path of the execution.
2. Generate an ECCT for the execution.
3. Compute global execution metrics.
4. Insert the execution's ECCT and metrics in a database.

Critical Path

TraceCompare computes the critical path of an execution using the algorithm proposed by our group and introduced in section 4.3.2. A graph of dependencies between threads is built while the trace is read. Then, the start and end points of an execution, both $(thread\ id, timestamp)$ pairs, are provided to the algorithm so that it can compute its critical path using the dependencies defined in the graph. The output of the algorithm is a list of segments belonging to the critical path of the execution, each characterized by a thread id, a thread status, a start timestamp and an end timestamp. The thread status takes one of the following values: running, preempted, interrupted, waiting for another thread or waiting for the operating system (block device, network, timer or user input).

The graph of dependencies between threads is built incrementally while the trace is read. Only nodes generated by events that occured during an execution are required to compute its critical path. Therefore, the critical path of an execution can be computed as soon as its end event is encountered (before the full trace has been read).

ECCT

The next step of the analysis is to generate the ECCT of the execution from the stacks that appear on each segment of its critical path. Backtracking to reread events of the trace, every time the end of an execution is encountered, would be highly inefficient. Instead of that, we generate a state history while we read the trace. Efficient State History Trees were proposed by our group and described in section 4.3.2. Using them for comparison of executions is an original contribution. When the time comes to generate the ECCT of an execution, the required information is obtained by querying the state history. The evolution of the following attributes is tracked in our state history:

— `threads/[tid]/cpu`: The CPU on which the thread `tid` is running.
— `threads/[tid]/stack`: The call stack of the thread `tid`.
— `cpus/[cpuid]/thread`: The thread running on the CPU `cpuid`.
— `blockdevice/threads`: Threads waiting for a block device.

The algorithm presented in Fig. 4.3 generates the ECCT of an execution from its critical path and a state history.

```
Input: critical path C, state history H, stack of threads T, ECCT E
                                  ▷ T and E are empty in the first recursive call to this function.
 1: for all e ∈ C do
 2:     UpdateThreadStack(H, T, e.tid, e.start)
 3:     if e.status = running then
 4:         stacks ← Query H for stacks of thread tid in
                time range [e.start, e.end].
 5:         for all s ∈ stacks do
 6:             if T.Size > 2 then
 7:                 s.val ← Concatenate( T[−2].stack, thread, s.val)
 8:             end if
 9:             E.Insert(s.val, s.duration)
10:         end for
11:     else if e.status = preempted then
12:         cpuId ← Query H for CPU of thread tid at time e.start.
13:         threads ← Query H for threads running on CPU cpuId in
                time range [e.start, e.end].
14:         for all t ∈ threads do
15:             e_preempt ← (t.val, running, t.start, t.end)
16:             GenerateECCT({e_preempt}, H, T, E)
17:         end for
18:     else if e.status = blockdevice then
19:         threads ← Query H for threads using a block device in
                time range [e.start, e.end].
20:         for all t ∈ threads do
21:             s ← Query H for stack of thread t.val at time t.start.
22:             s ← Concatenate(T[−1].stack, block device, s)
23:             E.Insert(s, e.duration/threads.size)
24:         end for
25:     else
26:         s ← Concatenate(T[−1].stack, e.status)
27:         E.Insert(s, e.duration)
28:     end if
29:     UpdateThreadStack(H, T, e.tid, e.end)
30: end for
```

Figure 4.3 Function GenerateECCT Generates the ECCT of an execution.

```
Input: state history H, stack of threads T, tid, ts
 1: for i ← 0 to T.Size − 1 do
 2:     if T[i].tid = tid then
 3:         T.Resize(i)
 4:     end if
 5: end for
 6: s ← Query H for stack of thread tid at time ts.
 7: if T.Size > 0 then
 8:     s.val ← Concatenate(threads[−1].stack, thread, s.val)
 9: end if
10: T.Push((tid, s.val))
```

Figure 4.4 Function UpdateThreadStack: Puts the enhanced stack of a thread at the top of a stack of threads.

Execution Metrics

It is convenient to be able to filter executions based on a wide range of metrics when performing a comparison. For example, we might want to compare executions that generated a lot of page faults against those that generated a few, or executions that allocated a lot of memory against those that allocated a few.

Depending on the events that are present in the source trace, we can keep track of the evolution of different metrics in the state history. Then, when we process an execution, we compute metrics for each segment of its critical path. The values of each segment are then combined into a global metric, on which filters can be applied in our comparison view.

Database

Once the ECCT and the global metrics of an execution have been computed, they are added to the executions database. An interesting observation is that there is no need to duplicate the structure of the ECCT for each execution. A generic ECCT can be formed by the union of all the other ECCTs. Its structure along with the function prototype of each node are stored once in the database. Each execution record can then refer to the nodes of the generic ECCT to specify metrics. Because the ECCTs issued from different executions of the same task normally have a similar structure, this original technique greatly reduces the size of the database.

Garbage collection

Two data structures are built incrementally while the trace is read: a graph of dependencies between threads and a state history tree. The size of both these data structures is proportional to the number of events read. For huge traces, storage of all this data might be a concern. Fortunately, the nodes added to the graph of dependencies, before the first event of an execution is read, are not needed to compute its critical path. Similarly, the intervals of the state history tree, that end before the begininnig of an execution, are not needed to compute its ECCT or global metrics. Therefore, whenever the available memory is getting low, it is possible to discard the parts of the data structures that won't be needed to process the pending executions. In addition to reducing memory usage, this strategy improves the time complexity of queries in both data structures. They become logarithmic with respect to the average size of an execution, instead of logarithmic with respect to the size of the analyzed trace.

4.4.4 Comparing Task Executions

The database of executions is used to populate a web-based view that allows interactive comparisons between groups of executions. The view is divided in three parts: the filters, the flame graph and the list of executions. An obvious benefit of providing a web-based view is that it facilitates collaboration. Anybody can easily share a link to its findings in a bug report, to help others fix the problem quickly. This is much more convenient than sharing huge trace files, let alone asking others to reproduce a complex bug themselves.

Filters

The aim of the filters is to allow the user to define two groups of executions to compare. They are presented as two columns of histograms. Each column is associated with a group of executions and each row is associated with a metric. Hence, each histogram shows the distribution of a metric within a given group. Initially, both columns are identical because all executions are included in both groups.

To filter the executions of a group, the user selects a region on an histogram [4]. For example, in Fig. 4.5, the left group contains executions with a total duration under 20 ms while the right group contains executions with a total duration above 50 ms. It is possible to set as many filters as desired.

Whenever a filter is added to a metric, the histograms of the other metrics are updated to only take into account the executions that belong to their group. This can reveal interesting correlations between metrics. For example, Fig. 4.5 shows a case where the CPU time of an execution is not correlated with its total duration, but is correlated with the number of bytes read from the disk.

Flame Graph

Differential flame graphs were introduced in section 4.3.3 as a way to compare two CPU profiles. The second part of our comparion view reuses this visualization tool, but with some adaptations.

First, our differential flame graphs are built from ECCTs instead of from CPU profiles. Fig. 4.6 shows that they can reveal latencies caused by chains of blocked threads and long disk requests. This is impossible with a simple CPU profile.

Second, our differential flame graphs are used to compare two groups of executions instead

4. The implementation is inspired from `http://square.github.io/crossfilter/`.

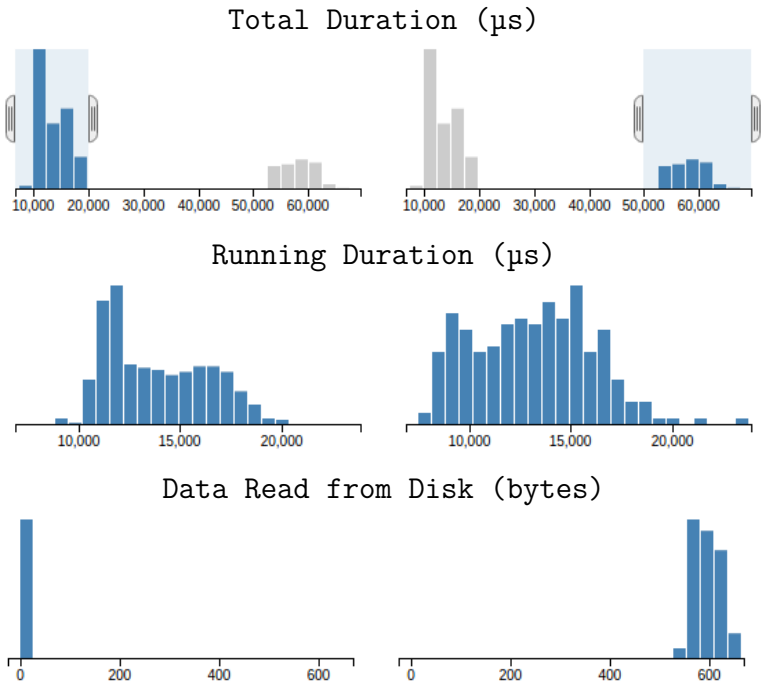


Figure 4.5 Comparison Filters

of two individual executions. The width of a box is proportional to the mean time spent in a calling context for executions of the right group. The hue of a box is computed from the number of standard deviations between mean times spent in a calling context in the two compared groups. This technique prevents colors to be applied to boxes in the presence of normal variations, which is a limitation of the original differential flame graphs.

Our differential flame graph is dynamically updated when filters are applied in the top part of the view.

Figure 4.6 Differential Flame Graph

List of executions

The bottom of the view presents two tables with the source trace and timestamp of a few executions sampled from the two compared groups. This data enables the user to find executions of interest in another viewer such as TraceCompass to analyze the precise ordering of events.

Filtering algorithm

TraceCompare uses a map-reduce algorithm to update histograms when filters are modified. "Map" functions are applied to all executions that are either added to or removed from a group following a filter modification. The functions determine in which bucket, of each histogram, affected executions belong. "Reduce" functions compute the new value of each histogram bucket.

A sorted index is kept for all metrics on which filters can be applied. Also, a filter is always specified as a range of accepted values for a given metric. Therefore, to update a filter, we simply find the first affected execution in the index and we traverse subsequent executions until we find the last affected one. This operation is performed in $O(logN + M)$ time, where N is the total number of executions and M the number of affected executions.

The user updates filters by dragging a handle on a histogram. Therefore, M is typically small at each mouse event and the views are updated quickly. The update takes a little bit more time when a lot of executions have a similar value for the filtered metric and are affected simultaneously. To keep the views responsive in that case, we can precompute the result of the reduction for segments of fixed size on each dimension.

TraceCompare uses the Crossfilter [5] implementation of this map-reduce algorithm. The algorithm has already been used to analyze multi-dimensional data sets of payment, meteorological and transportation data, but never to compare software performance metrics.

4.5 Case Studies

This section presents four performance problems, encountered in real applications, along with a description of how TraceCompare can diagnose them. The first two case studies were made on test applications that we built to closely reproduce problems found in enterprise software. The two subsequent case studies show how TraceCompare was used to find *real* performance

5. `https://github.com/square/crossfilter`

problems in MongoDB [6], an open-source database software.

4.5.1 CPU Contention in a Real-Time Application

Problem Summary

A realtime task is scheduled to be executed at a frequency of 100 Hz. Another task of higher priority (set with `nice`) is scheduled to be executed at a frequency of 30 Hz on the same CPU. At regular interval, the deadline of the second task occurs while the first task is running, increasing its execution time. Unfortunately, the developers are not aware of the existence of the second task.

Diagnosis

TraceCompare was used to compare slow executions of the first task against fast ones. The flame graph, partially reproduced in Fig. 4.7, immediatly revealed that the first task was preempted by the second during the slow executions. More interestingly, it showed call stacks from the second task. With these call stacks, it was clear where to look in the code to fix the problem. Without call stack events, the analysis tool could only have shown the name of the task that stole the CPU from the first task. Since threads are named after their parent, unless efforts are made to give them meaningful names, further investigation would have been needed to find out what was the purpose of the second task.

Figure 4.7 Differential Flame Graph Showing CPU Contention in a Real-Time Application

The histogram showing the "timestamp" metric of slow executions also provides useful information: it reveals that the problem is occurring at regular intervals.

6. `https://www.mongodb.org`

4.5.2 Disk Contention in a Server Application

Problem Summary

A server has to read data from the disk to fulfill requests. At regular intervals, a request is about 5 times slower than usual.

Diagnosis

TraceCompare revealed that the additional latency found in the slow requests came from abnormaly long `read()` system calls. With a tool whose scope is limited to a single process, it would have been hard to pursue the analysis further. Fortunately, TraceCompare was able to leverage the information from a trace of the whole system to show precisely which other threads used the disk during these long system calls and what were their call stacks.

TraceCompare highlighted a single thread that always used the disk during the slow executions, but never during the fast executions. That thread was performing an `fsync()` system call after having written to a log file. With this information, fixing the problem was straightforward.

4.5.3 Lock Contention in MongoDB

Problem Summary

A client application generates data and inserts it into a MongoDB database (version 2.5.4). Most of the time, the whole operation takes around 10 ms. However, a fraction of the time, the operation takes more than 100 ms. Data is generated at less than 1 MB/s.

Diagnosis

The database insertions were performed serially, so it was unlikely that the performance variation was due to CPU or lock contention. Also, the MongoDB's documentation states that insert operations don't wait for data to be committed on disk before returning (a dedicated command is available to wait for that). Therefore, we didn't expect disk latency to affect the performance of our application. Note that if our application had produced data at a higher rate, it would have been understandable that MongoDB throttled new insertions to flush its memory buffers.

The differential flame graph of TraceCompare, partially reproduced in Fig. 4.8, revealed that slow executions had waited for a mutex protecting access to a list of pending changes.

During these wait times, the mutex was held by a journalization thread. This information alone doesn't lead to an obvious fix and could have been found by lock analysis with a tool such as Intel® VTune™ [7]. However, TraceCompare sets itself apart from other tools by also revealing a function running on the journalization thread for almost all of the wait time. We analyzed the code of this function and discovered that it was not using the list protected by the mutex. Therefore, we wrote a simple patch to release the mutex before calling the costly function. We verified that the patch correctly fixed the serious performance problem that we had uncovered and we submitted it to the MongoDB developers. All this was done without any prior knowledge of the MongoDB codebase.

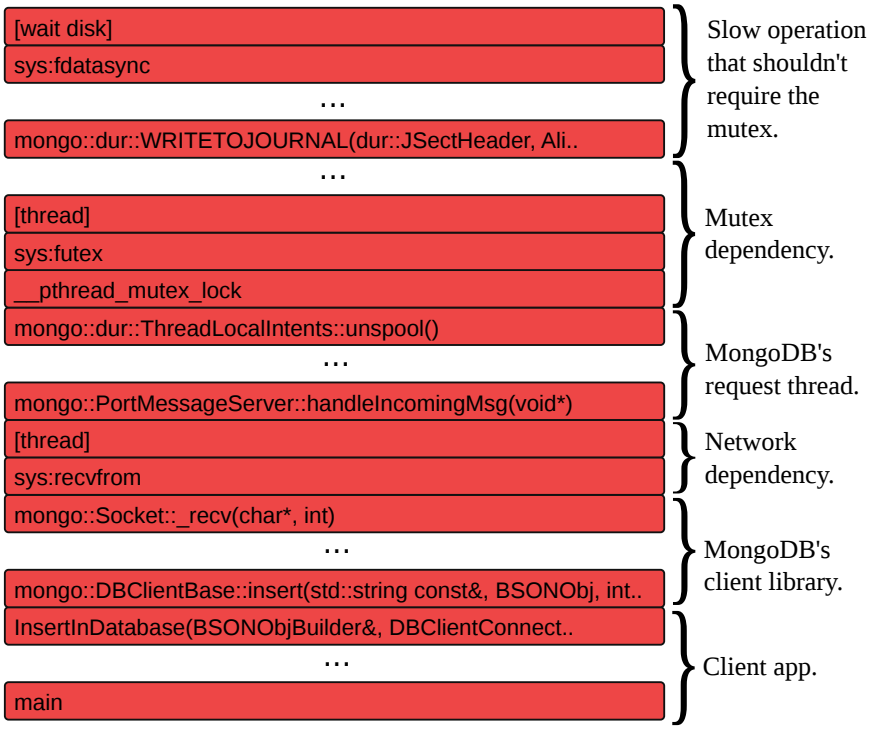


Figure 4.8 Differential Flame Graph Showing a Lock Contention in MongoDB

This diagnosis was made with an unmodified MongoDB binary. We only inserted probes in our client application to delimit the task to analyze. To track the latency, from our client application to the journalization thread of MongoDB, TraceCompare had to match TCP packets exchanged between both applications, and to analyze wake-up events within the Linux kernel.

7. `https://software.intel.com/intel-vtune-amplifier-xe`

4.5.4 Sleep in MongoDB

Problem Summary

Batch insert commands are sent to a MongoDB server (version 3.0.0 rc 10, WiredTiger storage engine). The commands are run in less than 700 µs most of the time. However, about 1 in 10 000 commands takes between 3 and 5 seconds to complete.

Diagnosis

TraceCompare made it easy to compare a group of fast executions against a group of slow executions. The comparison revealed that there was in fact two distinct sources of latency in our test scenario. Some executions were waiting for the disk within `pwrite64()` system calls while other were blocked by a timer. Since disk contention can be expected when a lot of data is inserted in a database, we decided to create a second filter to focus on the more suspicious timer contention.

The flame graph showed that the timer contention was caused by a sleep within a function responsible to obtain a hazard pointer to a page of data. We located the sleep call in the source code. It was accompagnied by a comment explaining that if another thread was trying to evict the page from the cache at the same time, it was necessary to wait a while before retrying to obtain the pointer.

TraceCompare didn't show any dependant work being done during the long sleeps. In fact, we were able to confirm that both the CPU and disk activity were very low during the sleeps, by using the information provided in the tables at the bottom of the view to locate the slow executions in another trace analysis tool. At that point, we were confident that we had found a synchronization bug in MongoDB, along with a precise diagnostic.

Despite the fact that the sleeps caused huge delays, they didn't account for a significant proportion of the trace. They wouldn't have caught our attention if we had used a profiler, which only shows global statistics. Having the ability to create groups of executions using multiple custom filters was a key feature to diagnose this problem. Also, having a trace of timestamped events allowed us to obtain resource usage statistics for specific time ranges.

4.6 Performance Analysis

We conducted experiments to evaluate the tracing overhead of our solution as well as the performance of the analysis. It is crucial to be able to record all the events required by TraceCompare with low overhead. Indeed, it is unlikely that our solution would be used on

production systems if it decreased performance noticeably. Rare bugs that occur under very specific conditions would therefore be out of its reach. Also, having a low overhead ensures that the behavior captured in the trace is similar to that of the uninstrumented system. On the other hand, a fast analysis time helps system administrators respond to problems quickly.

4.6.1 Environment

All experiments were run on a computer with a quad-core Intel® Core™ i7-3770 CPU running at 3.4 GHz, 16 GB of DDR3 memory and a 7200 RPM hard drive. The frequency scaling governor of the CPU was set to "performance" to ensure that it always runs at its maximum frequency. The Linux kernel version was 3.13.0-49 while the LTTng version was 2.6.0.

4.6.2 Cost of Tracing

We wanted to measure the overhead incurred by each step of the generation of our new `syscall_stack` event. To do so, we first created a microbenchmark that invokes the `getpid()` system call 100 million times and we timed its completion time when different parts of our kernel module were enabled. We repeated each experiment 20 times and always got a standard deviation of less than 0.5%. The `getpid()` system call was chosen because of the low variability of its duration.

Registering empty probes to tracepoints located at the entry and exit of system calls incurs a cost of 96 ns per system call. This is because the Linux kernel has to go through a slow path whenever there are probes registered to system call tracepoints. Adding our code to detect long system calls in the probes brings an additionnal cost of 42 ns. These costs apply to all system calls, even those that turn out to be faster than the threshold for long system calls. When a long system call is detected, a signal is sent to the application so that it captures its call stack. Sending the signal and executing an empty signal handler takes 1.2 µs.

Next, we microbenchmarked the operations executed within the signal handler. We used microbenchmarks that repeat the measured operation 100 million times. We repeated each experiment 20 times and always got a standard deviation of less than 2%.

The total time to unwind the stack depends on the number of frames currently on the stack. If it's the first time that a stack frame is encountered, it takes 326 ns to process it. This is 20% faster than without our custom optimizations. Because decoded unwinding rules are cached, subsequent processing of the same stack frame takes only 7 ns. There is also a base cost between 30 ns and 700 ns, depending on whether all stack frames are handled by cached rules. Once a stack has been captured, it is written in an LTTng buffer, an operation that

requires 175 ns.

`cpu_stack` events are generated from a signal sent at the expiration of a timer. Enabling the timer has a negligible overhead. The execution of the signal handler takes the same time as for `syscall_stack` events.

Finally, we wanted to determine how our instrumentation affects typical applications. Table 4.1 shows the time required to run 4 programs under different tracing scenarios. The overhead in percentage is reported in Table 4.2. Each test case was run 100 times.

The "Traditional tracing of system calls" scenario is a baseline to compare with the "Tracing stack events" scenario. Indeed, our custom `syscall_stack` event reveals long system calls with a significatively lower cost than the traditionnal system call events. TraceCompare requires a trace recorded under the "Tracing stack and critical path events" scenario.

Table 4.1 Execution Time of Test Programs (in seconds)

	prime		babeltrace		find		mongod	
	Mean	Dev.	Mean	Dev.	Mean	Dev.	Mean	Dev.
Base	9.11	0.00	17.57	0.17	14.39	0.18	9.91	0.28
Traditionnal tracing of system calls	9.11	0.00	17.81	0.14	14.99	0.28	10.72	0.06
Tracing stack events	9.12	0.00	17.74	0.15	14.67	0.35	10.09	0.20
Tracing stack and critical path events	9.13	0.00	17.76	0.15	15.08	0.30	10.76	0.05

Table 4.2 Overhead of Tracing

	Overhead (%)			
	prime	babeltrace	find	mongod
Traditionnal tracing of system calls	0.0	1.4	4.1	8.2
Tracing stack events	0.1	1.0	1.9	1.8
Tracing stack and critical path events	0.2	1.1	4.8	8.6

`prime` is an application that computes a list of prime numbers up to 10 000. It does no system calls, so the overhead is mostly due to the statistical CPU profiling. `babeltrace` is a single-threaded program that reads CTF traces. It does a lot of system calls to open trace files and output results, but few of them are long enough to trigger the generation of a stack event. The 1% overhead is mostly due to the kernel module tracking the duration of each system call. `find` is a command-line tool that searches files recursively in the file system. It spends most of its time blocked on block device requests. Each request generates interrupts and context switches, which contribute to the 5% overhead of tracing critical path events. `mongod` is an open-source database server. It was tested using the client application

presented in section 4.5.3. The interactions between multiple threads through synchronization primitives and TCP packets explain the 9% overhead.

4.6.3 Cost of the Analysis

The time required to build the database of executions for each of the case studies presented earlier is reported in Table 4.3. This metric is correlated with the size of the trace rather than its duration. It is always less than 3 times the time required to read the trace with `babeltrace`[8] (a tool that just decodes the events of a trace).

The table also shows that the size of the generated database is on average 10% of the size of the source trace. Yet, it contained enough information to perform the precise diagnosis that we presented earlier. By keeping a database of executions rather than voluminous traces, system administrators could save storage space without sacrificing the convenience of being able to analyze the behavior of a system over a long period of time.

Table 4.3 Cost of Building the Database of Executions

	Trace		Database	
	Recording Time	Size (MB)	Build Time	Size (MB)
4.5.1	13 min 26 s	52	17 s	7
4.5.2	22 min 42 s	86	20 s	7
4.5.3	2 min 38 s	31	7 s	3
4.5.4	5 min 11 s	129	32 s	13

Users can dig into a database of executions using the Web interface of TraceCompare, which provides responsive filters. Table 4.4 presents the time required to refresh the views when different filters are created. We used a database of 200 000 executions, each associated with 15 metrics. For each test case, the user created a new filter by selecting a 10 pixels wide region on a histogram. In test case (a), the histogram presented a uniformly distributed metric. The selected region contained 4000 executions. In test case (b), the histogram presented a metric with a prominent mode and a few outliers. The selected region contained 195 000 executions. Test case (c) was identical to test case (b), except that the result of the reduction had been precomputed for the selected range (see section 4.4.4).

The results reveal that precomputation is required in order to be able to refresh the views at 30 fps. If the user prefers to avoid the precomputation time, a refresh rate of 13 fps can be obtained, provided that the filtered metric is uniformly distributed.

8. `http://git.efficios.com/?p=babeltrace.git`

Table 4.4 Cost of Refreshing the Web Interface

Test Case	Time (ms)	
	Mean	Std. Dev.
(a) Uniform Distribution	75.9	1.1
(b) Mode and Outliers	1640.7	6.3
(c) Mode and Outliers, Precomputed	3.4	0.3

4.7 Future Work

Our comparison tool has hard-coded rules to identify latencies caused by OS level synchronization between threads (mutex, pipe, signal, TCP packet, etc.), or by CPU or disk contention. Its scope could be widened by adding logic to account for GPU contention, CPU contention across virtual machines, or userspace-only synchronization through standard libraries. It is however impossible to handle out of the box the specificities of each application (e.g. custom task queues, custom IPC through shared memory, etc.). The declarative language used for trace analysis in [31] could be extended to allow the expression of these specificities.

We used TraceCompare to analyze performance regressions between multiple versions of the same program. Renamed functions can easily be dealt with by allowing the user to provide a file, mapping old function names to new names. However, the biggest challenge comes from major code refactoring, during which functions can be splitted or merged. Static analysis techniques exist to identify cloned syntactic blocks between multiple source files [86]. Techniques also exist to retrieve a mixed stack of function calls and syntactic blocks for a thread [87]. These techniques could be combined to allow a meaningful comparison between traces recorded on different versions of the same program, even after a major refactoring.

Our userspace tracing library can only capture the stack for ELF binaries. It should be extended to properly handle programs written in a dynamic language and JIT compiled.

It would also be useful to allow userspace processes to associate metrics with the current execution (e.g. number of returned results). Filters on these metrics could then be used to specify the execution groups to compare with more granularity.

4.8 Conclusion

In this paper, we presented a new tool that facilitates the detection and diagnosis of performance variations between multiple executions of the same task. A novel technique was proposed to capture the userspace stack of programs at key moments. The stacks were

recorded along with kernel events using the LTTng tracer. We showed how to combine all these events to generate a database of ECCTs, a new data structure to describe latency that occurs at multiple levels during task executions. Then, we introduced an intuitive GUI that allows a user to specify groups of executions, using custom filters, in order to visualize their differences. An efficient map-reduce algorithm makes the views responsive for typical use cases, and userspace stacks make it easy to find the source code associated with identified differences. We detailed how our tool was able to discover performance bugs commonly found in enterprise software and in MongoDB, a popular open-source software. Finally, we showed that the overhead of tracing the events, required by our comparison analysis, is always under 9%, which means that our solution can be used on production systems.

CHAPITRE 5 DISCUSSION GÉNÉRALE

5.1 Analyse statistique des tests de performance

À la section 4.6.2, nous avons évalué le surcoût de notre solution pour capturer la pile d'appels des longs appels système. La durée de chaque opération a été évaluée en faisant la différence entre les temps moyens pris par deux cas de tests (avec et sans l'opération dont on souhaite obtenir la durée). Nous présentons ici des intervalles de confiance à 95% pour toutes ces différences entre moyennes. Aussi, nous présentons la valeur-p de tests d'hypothèse vérifiant si ces différences pourraient être dues au hasard (seuil de signification de 5%).

Le tableau 5.1 présente les résultats pour différentes parties du module noyau surveillant la durée des appels système. Le tableau 5.2 présente les résultats pour la génération d'un événement avec LTTng-UST.

Tableau 5.1 Intervalles de confiance à 95% et valeurs-p pour le surcoût de différentes parties du module noyau surveillant la durée des appels système (en nanosecondes)

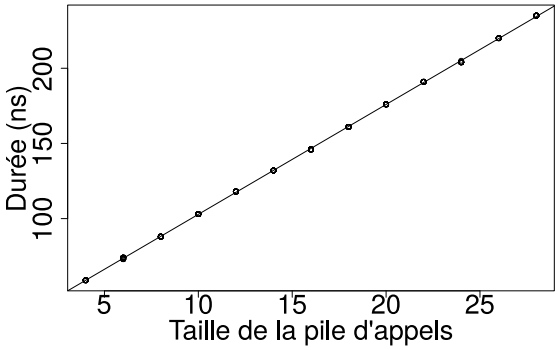
Cas de test	Temps		Surcoût	
	Moyenne	Écart-type	Intervalle de confiance	Valeur-p
Base	37.87	0.14	–	–
Sondes vides	134.22	0.06	[96.28, 96.40]	0
Suivi de la durée	176.48	0.09	[42.22, 42.30]	0
Envoi d'un signal	1345.68	0.39	[1169.05, 1169.36]	0

Tableau 5.2 Intervalle de confiance à 95% et valeurs-p pour le surcoût de la génération d'un événement avec LTTng-UST (en nanosecondes)

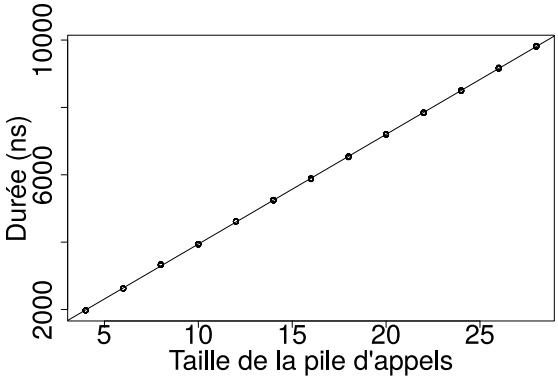
Cas de test	Temps		Surcoût	
	Moyenne	Écart-type	Intervalle de confiance	Valeur-p
Base	0.26	0.00	–	–
Génération d'un événement	174.93	2.36	[173.76, 175.59]	0

La taille de chaque échantillon est 20. Les échantillons respectent les conditions nécessaires pour le calcul d'un intervalle de confiance pour la différence entre deux moyennes : distribution normale, pas de différence majeure entre les écarts-types des valeurs comparées et mesures indépendantes. Puisqu'aucun intervalle de confiance n'inclut zéro et que toutes les valeurs-p sont nettement inférieures au seuil de signification, on peut rejeter l'hypothèse que les différences calculées sont dues au hasard.

L'article discute aussi du temps requis pour capturer une pile d'appels avec `libunwind`. Étant donné que la durée de cette opération varie selon le nombre d'étages de la pile d'appels, nous avons répété nos expériences avec plusieurs tailles de piles d'appels. Nous avons ensuite effectué des régressions linéaires avec les résultats de ces expériences. Sur la figure 5.1, on voit que les droites obtenues par régression linéaire épousent bien les données obtenues expérimentalement. Le coefficient de détermination R^2 est 0,99992 pour le cas avec cache (a) et 0,99995 pour le cas sans cache (b), ce qui confirme que le modèle linéaire est bien adapté aux données.



(a) Avec les règles de restauration des registres en cache.



(b) Sans cache.

Figure 5.1 Durée de la capture d'une pile d'appels en fonction de sa taille.

Bref, les analyses de cette section nous confirment que les résultats présentés dans l'article sont statistiquement significatifs.

5.2 Surcoût du traçage sur Mac OS X et Windows

Le traceur LTTng ne permettait pas de capturer la pile d'appels d'un processus en espace utilisateur avant l'arrivée de notre solution. Cette fonctionnalité était cependant déjà offerte par d'autres traceurs.

Sur Windows, le traceur ETW permet de capturer une pile d'appels lors de l'occurrence de certains événements du noyau. On peut choisir pour quels types d'événements activer cette fonctionnalité lors du démarrage d'une session de traçage. Il n'est toutefois pas possible de réserver cette action aux événements plus longs qu'un seuil, comme le permet notre solution.

Sur Solaris, FreeBSD et Mac OS X, le traceur DTrace capture une pile d'appels lorsqu'il rencontre l'instruction `ustack()` dans le script D associé à un point de trace. Le langage D offre une grande flexibilité pour spécifier quand effectuer cette action.

Nous avons effectué des tests pour comparer le coût de ces différentes solutions. Tous les tests de cette section ont été effectués sur un MacBook Pro 2012 équipé d'un processeur Intel Core i7-3720QM cadencé à 2.6 GHz, de 8 Go de mémoire DDR3 et d'un disque SSD. Les solutions étudiées sont :

— LTTng 2.6.0 sur Ubuntu 14.04, noyau 3.13.0
— DTrace sur Mac OS X 10.9.5
— ETW sur Windows 8.1

D'abord, nous avons évalué le temps requis pour enregistrer la pile d'appels d'un appel système à l'aide de chaque solution. Les résultats sont présentés dans le tableau 5.3. Pour évaluer LTTng et DTrace, nous avons utilisé un programme invoquant à plusieurs reprises l'appel système `getpid()` à partir d'une pile de 10 éléments. Ces deux traceurs permettent de réserver la capture de la pile aux appels système plus longs qu'un seuil. Nous présentons donc le coût pour un appel plus court que le seuil (pas de capture de la pile) et un appel plus long que le seuil (avec capture de la pile). Pour ETW, nous avons remplacé l'appel système `getpid()` par `NtClose()`. Ce traceur ne permet pas de définir un seuil pour capturer la pile d'appels. Chaque test a été répété 20 fois.

Tableau 5.3 Surcoût de la capture de la pile d'appels d'un appel système dans des environnements divers

	Appel court (µs)	Appel long (µs)
LTTng sur Linux	0.80	2.54
DTrace sur Mac OS X	1.87	3.15
ETW sur Windows	n/a	2.02

Notre solution se compare avantageusement à DTrace, dont le langage dynamique n'est pas

aussi efficace que notre code natif pour surveiller la durée des appels système. Notons aussi que nous n'avons pas été capables d'exécuter un programme de test faisant 100 millions d'appels à `getpid()` sans que DTrace ne perde des événements, et ce malgré le fait que nous utilisions la valeur maximale permise pour la taille des tampons. Cela est certainement dû au fait que DTrace produit des traces au format textuel.

ETW a un coût légèrement plus faible que celui de notre solution. Ce traceur capture la pile d'appels à partir du noyau, ce qui lui évite l'envoi coûteux d'un signal à l'espace utilisateur. Il serait intéressant de mesurer le coût de notre solution si `libunwind` était exécuté dans le noyau. Notons que cela nécessiterait l'exécution du code à octets de la section `.eh_frame` des binaires ELF dans le noyau, une opération risquée. Aussi, il serait plus difficile de supporter une grande variété de langages dynamiques avec cette technique qu'avec la technique actuelle.

Nous avons aussi évalué l'impact du traçage des événements requis par notre analyse sur `prime` et `mongod`, deux applications introduites à la section 4.6.2. Les résultats sont présentés dans le tableau 5.4. Les surcoûts en pourcentage sont présentés dans le tableau 5.5. Chaque test a été répété 100 fois.

Le script utilisé pour capturer les événements requis avec DTrace est reproduit à l'annexe A. Pour ETW, nous avons capturé les piles d'appels lors des changements de contexte plutôt que lors des appels système. Cela pallie au fait que ce traceur ne permet pas de réserver cette action aux appels système les plus longs. Aussi, avec ce traceur, les événements sont activés par groupe. Nous n'avons donc pas pu tester le cas où seuls les événements de piles d'appels sont activés. Enfin, nous n'avons pas trouvé d'équivalent à l'événement `sched_wakeup` sur cette plate-forme. Cet événement est crucial pour repérer les dépendances entre fils d'exécution lors du calcul du chemin critique d'une tâche.

Tableau 5.4 Temps d'exécution de programmes de tests dans des environnements divers (en secondes)

Environnement	Événements tracés	prime		mongod	
		Moyenne	Écart-type	Moyenne	Écart-type
LTTng sur Linux	Aucun	11.8	0.2	13.0	0.7
	Piles	11.8	0.1	13.5	0.6
	Piles et chemin critique	11.8	0.1	14.0	0.8
DTrace sur Mac OS X	Aucun	12.2	0.0	17.9	0.1
	Piles	12.4	0.0	18.7	0.0
	Piles et chemin critique	12.4	0.0	22.2	0.2
ETW sur Windows	Aucun	10.9	0.0	30.2	1.3
	Piles et chemin critique	10.9	0.0	37.5	0.9

Ces résultats révèlent que notre solution complète a un impact plus faible que les autres sur les applications testées. Il est surprenant de constater que DTrace impose un surcoût de 1%

Tableau 5.5 Surcoût du traçage dans des environnements divers

Environnement	Événements tracés	Surcoût (%)	
		prime	mongod
LTTng sur Linux	Piles	0.2	3.5
	Piles et chemin critique	-0.1	7.6
DTrace sur Mac OS X	Piles	1.1	4.6
	Piles et chemin critique	1.0	24.0
ETW sur Windows	Piles et chemin critique	0.0	24.0

à l'application `prime`. Puisque `prime` ne fait pas d'appels système, ce surcoût est uniquement dû au profilage statistique. Cette opération est effectuée avec un surcoût négligeable par de nombreux autres outils. La valeur élevée peut être expliquée par le fait que DTrace symbolise les piles d'appels avant de les enregistrer, tandis que cette opération doit être faite hors ligne avec LTTng et ETW. ETW impose un surcoût significativement plus élevé que LTTng pour enregistrer les événements du noyau nécessaires au calcul du chemin critique. Cela met en évidence les multiples optimisations présentes dans LTTng pour lui permettre d'enregistrer des événements du noyau en seulement 600 cycles dans des systèmes parallèles [38]. En comparaison, Microsoft annonce qu'ETW prend de 1500 à 2000 cycles pour faire le même travail [88].

Les résultats pour `prime` comportent deux valeurs qui peuvent sembler aberrantes. Cependant, nous avons obtenu des valeurs similaires chaque fois que nous avons reproduit l'expérience. Les valeurs pourraient donc être expliqués par des facteurs tels qu'une utilisation différente de la mémoire ou de l'ordonnanceur.

Le fait que nous soyons parvenus à tracer la plupart des événements nécessaires à notre analyse sur trois plates-formes différentes est un résultat intéressant en soi. Cela signifie que des développeurs pourraient avoir recours à une solution unique pour analyser la performance de leurs applications sur toutes les plates-formes qu'ils supportent, réduisant ainsi le temps consacré à apprendre le fonctionnement de nouveaux outils. Ce besoin avait été identifié à la section 3.1.4. L'absence d'équivalent à l'événement `sched_wakeup` sur Windows pourrait être palliée par de l'instrumentation en espace utilisateur.

5.3 Visualisation simultanée de multiples traces d'exécution

TraceCompare révèle aisément les sources de latence au sein de multiples exécutions d'une tâche et facilite la comparaison entre des groupes d'exécutions. Afin de simplifier l'analyse, l'outil ne tient pas compte de l'ordre des événements. Cette information peut toutefois être importante pour comprendre les problèmes de performance affectant un système parallèle.

Pour cette raison, TraceCompare fournit les détails nécessaires pour retrouver des exécutions individuelles dans un autre outil. On peut ainsi analyser la chronologie des événements survenus durant à une exécution dans un outil tel que TraceCompass. Malheureusement, cet outil ne permet pas de visualiser plusieurs exécutions simultanément. Nous avons donc imaginé un mode de comparaison pour TraceCompass, que nous avons partiellement prototypé.

Dans ce nouveau mode de comparaison, l'utilisateur spécifie deux exécutions à comparer. Les analyses déjà disponibles dans TraceCompass sont alors montrées simultanément pour chaque exécution. Lorsque l'utilisateur sélectionne un intervalle de temps sur une vue montrant une exécution, l'intervalle correspondant est automatiquement sélectionné sur les vues de l'autre exécution. Il est alors facile de comparer les chronologies d'événements des deux exécutions autour de l'intervalle sélectionné.

Nous proposons une nouvelle technique pour trouver efficacement les temps correspondants entre deux exécutions. Cette technique s'appuie sur des arbres d'appels améliorés. Tout comme les ECCTs, ces arbres sont construits en superposant les piles d'appels des fils d'exécution appartenant au chemin critique d'une exécution. Cependant, les appels distincts à une même fonction engendrent des nœuds distincts et les nœuds sont ordonnés. Aussi, il est possible d'intégrer des événements ponctuels (par exemple, l'acquisition d'un verrou) comme feuilles de ces arbres.

Pour trouver les temps correspondants entre deux exécutions, nous trouvons les nœuds correspondants dans leurs arbres d'appels améliorés. Pour ce faire, nous parcourons les arbres par niveaux. Les nœuds à la racine de chaque arbre sont d'abord mis en correspondance. Puis, les enfants des nœuds correspondants sont à leur tour mis en correspondance. Cette procédure est répétée jusqu'aux nœuds feuilles.

Nous avons observé que les enfants d'un nœud étaient fréquemment trop nombreux pour que l'on puisse les aligner avec ceux d'un autre nœud par un algorithme de programmation dynamique. Le temps d'exécution quadratique d'un tel algorithme est en effet prohibitif. Nous avons donc recours à une heuristique dont les étapes sont illustrées à la figure 5.2. Premièrement, nous utilisons une fenêtre glissante pour identifier les sous-séquences répétées. Ces répétitions sont remplacées par un nœud temporaire (a). Deuxièmement, nous mettons en correspondance les nœuds qui se retrouvent exactement une fois dans chaque séquence (b). Dans nos expérimentations, ces deux étapes permettaient de traiter la majorité des nœuds. Les nœuds restants, moins nombreux que dans les séquences initiales, sont alignés par un algorithme de programmation dynamique (c). Enfin, les sous-séquences répétées sont réintégrées et mises en correspondance par un algorithme vorace (d).

Il est possible qu'aucune des stratégies destinées à réduire la taille des séquences mises en

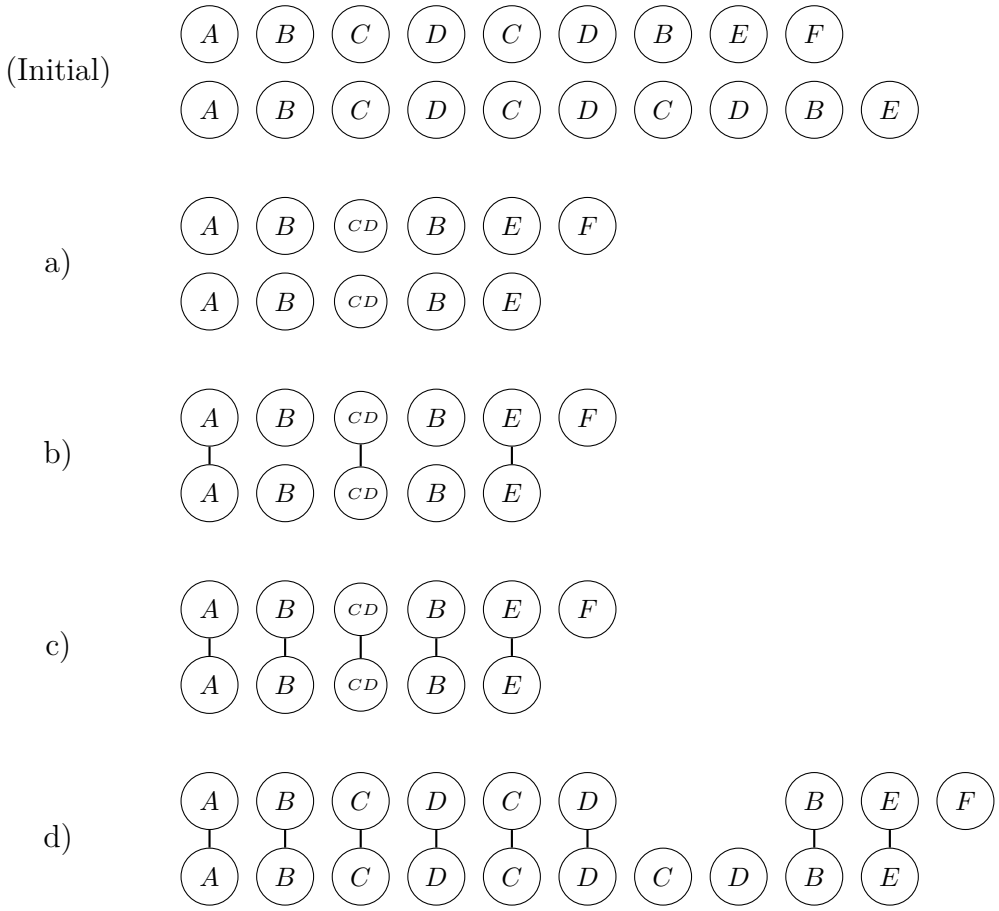


Figure 5.2 Heuristique d'alignement de séquences

correspondance ne s'applique pour certaines traces. Dans de tels cas, la complexité de notre solution est dominée par l'exécution de l'algorithme de programmation dynamique : $O(nm)$, où n et m correspondent au nombre d'événements dans chacune des deux traces comparées. Heureusement, lors de tests de notre prototype avec des traces du démarrage de Chrome, de l'installation d'un programme avec `apt-get` et du traitement de requêtes Web par un serveur Apache, nous avons constaté que les règles de notre heuristique produisaient le résultat souhaité. Les traces étaient découpées en plusieurs courtes sous-séquences. L'unique exécution de l'algorithme de programmation dynamique en $O(nm)$ était ainsi remplacée par plusieurs exécutions en $O(xy)$, où x et y sont plus petits que n et m de plusieurs ordres de grandeur. Néanmoins, le prototypage de cet outil nous a permis de constater qu'il était beaucoup moins efficace que TraceCompare pour détecter et diagnostiquer des problèmes de performance.

5.4 Respect des contraintes de l'industrie

À la section 3.1, nous avons présenté des contraintes présentes dans l'industrie des applications de haute performance. Nous revenons maintenant sur ces contraintes en expliquant comment elles sont gérées par TraceCompare.

5.4.1 Les systèmes analysés sont méconnus des développeurs.

La comparaison entre de multiples exécutions d'une même tâche permet à TraceCompare de repérer automatiquement des problèmes de performance. Grâce à la capture de piles d'appels à des moments clés, l'outil révèle aussi le code fautif associé à ces problèmes. Nous avons détecté, diagnostiqué et corrigé des problèmes de performance importants dans plusieurs logiciels dont nous n'avions aucune connaissance préalable grâce à TraceCompare. Notre solution est donc bien adaptée à l'analyse de systèmes méconnus.

5.4.2 Les systèmes analysés comportent des interactions entre de multiples composants.

TraceCompare retrouve tous les fils d'exécution affectant le temps de complétion d'une exécution grâce à un algorithme de calcul du chemin critique, et ce même lorsqu'ils sont répartis sur plusieurs machines. L'outil tient aussi compte des interactions avec le système d'exploitation.

5.4.3 Les systèmes à analyser sont victimes de problèmes difficiles à reproduire.

TraceCompare n'a pas recours à des métriques globales qui masquent les problèmes intermittents. Son interface graphique montre la distribution complète d'une grande variété de métriques et simplifie la comparaison des exécutions ayant des valeurs extrêmes avec celles ayant des valeurs normales. De plus, nous avons vu que le traçage des événements requis par TraceCompare a un surcoût inférieur à 9%. Ainsi, notre solution peut être utilisée pour diagnostiquer des problèmes qui se produisent uniquement dans un environnement de production.

5.4.4 Les systèmes à analyser s'exécutent sur des plates-formes diversifiées.

Nous avons discuté à la section 5.2 de l'adaptation de notre solution pour les environnements Mac et Windows.

5.4.5 Les développeurs ont peu de temps à consacrer à l'analyse de performance.

Pour les deux études de cas sur MongoDB, nous sommes parvenus à bien comprendre les problèmes découverts en moins d'une heure une fois les traces d'exécution collectées. Cela est impressionnant sachant que nous n'avions aucune connaissance préalable du code source de ce logiciel. Pour repérer les problèmes de ces deux études de cas, nous avons eu recours à un script qui trace l'envoi de charges variées à la base de données et qui soumet automatiquement les traces enregistrées à TraceCompare. Ce traitement par lots réduit le temps humain devant être consacré à l'analyse de la performance. Notons aussi que TraceCompare regroupe l'analyse d'une grande variété de problèmes au sein d'un seul outil.

CHAPITRE 6 CONCLUSION

Nous concluons ce mémoire en faisant une synthèse des contributions que nous avons apportées au domaine de l'analyse de performance dans les systèmes parallèles. Nous présentons aussi les limitations de notre solution et recommandons des améliorations futures.

6.1 Synthèse des travaux

6.1.1 Atteinte des objectifs

Ce travail de recherche visait à déterminer s'il était possible d'accélérer le diagnostic de variations de performance dans un système informatique en ayant recours à un algorithme qui identifie automatiquement les différences entre deux groupes de traces d'exécution. La solution que nous avons proposée confirme que c'est possible. En effet, nous avons mené quatre études de cas dans lesquelles TraceCompare a permis un diagnostic rapide et exact de problèmes de performance majeurs dans des logiciels libres et d'entreprises.

Nous avons aussi atteint nos 5 objectifs spécifiques. (1) Nous avons développé une solution pour capturer la pile d'appels d'applications en espace utilisateur à des moments-clés, faisant ainsi le lien entre des événements de bas niveau et la logique applicative. (2) Nous avons montré qu'un algorithme de calcul du chemin critique permettait de retrouver tous les événements affectant le temps de complétion d'une exécution donnée. (3) Nous avons conçu un algorithme utilisant des ECCTs pour extraire les différences significatives entre des groupes d'exécutions. (4) Nous avons conçu une interface Web permettant de spécifier des groupes d'exécutions et de visualiser leur différences à l'aide d'histogrammes et de *flame graphs*. (5) Nous avons présenté quatre études de cas démontrant l'efficacité de notre solution.

6.1.2 Contributions scientifiques

Notre solution comporte plusieurs contributions scientifiques. D'abord, nous avons introduit le ECCT, une nouvelle structure de données décrivant précisément les latences survenues au cours de l'exécution d'une tâche. Ces latences peuvent être causées par une forte utilisation du processeur par une fonction de l'espace utilisateur, par de l'attente entre des fils d'exécution, par le système d'exploitation, par le matériel, etc. Un ECCT permet de connaître le code applicatif associé à chacune de ces latences, ce qui facilite la correction des problèmes. Nous avons aussi vu qu'une base de données de ECCTs construite à partir d'une trace d'exécution est moins volumineuse que la trace elle-même. La base de données contient tout de même

suffisamment d'information pour permettre le diagnostic précis de problèmes de performance.

L'application d'un algorithme de *map-reduce* pour filtrer et grouper des exécutions selon leurs caractéristiques de performance constitue aussi une contribution scientifique. Grâce à cette stratégie, nous avons pu proposer une vue novatrice permettant de construire deux groupes d'exécutions à l'aide de filtres et de visualiser leurs différences instantanément. Les différences sont révélées à l'aide d'histogrammes montrant la distribution de différentes métriques et à l'aide de *flame graphs* différentiels.

Enfin, nous avons proposé une technique pour tracer uniquement les appels système dont la durée dépasse un seuil. Cette technique a un surcoût deux fois moins élevé que le traçage traditionnel de tous les appels systèmes. Nous utilisons une table de hachage RCU pour garantir une extensibilité à plusieurs cœurs.

6.2 Limitations de la solution proposée

Une limitation de ce travail de recherche est que nous avons mené nos tests sur un nombre limité d'applications. Nous avons visité plusieurs entreprises et scruté les répertoires de bogues de plusieurs logiciels libres pour nous assurer que notre solution était adaptée aux besoins de l'industrie. Nous avons aussi obtenu des résultats concluants lors de plusieurs études de cas. Cependant, il serait important de mettre notre solution entre les mains d'un grand nombre d'utilisateurs externes afin d'obtenir davantage de rétroaction. Le fait que TraceCompare soit distribué sous une licence libre et que notre laboratoire collabore avec plusieurs entreprises permettra sans doute d'obtenir cette rétroaction lors des prochains mois. Il serait particulièrement intéressant de savoir comment se comporte notre solution dans des environnements virtualisés tels que OpenStack ou Amazon Web Services.

Une autre limitation de nos travaux est que nous avons toujours configuré notre traceur pour écrire les événements en continu sur le disque. Les traces générées par une telle technique deviennent rapidement très volumineuses. Il est impensable que des entreprises exploitant de grands centres de données puissent gérer le stockage de traces générées ainsi. Il serait donc intéressant d'utiliser TraceCompare pour analyser des traces générées avec le mode *flight recorder* de LTTng. Ce mode permet d'enregistrer des événements dans des tampons circulaires en mémoire et de les copier sur le disque sur demande. Des conditions pourraient être définies pour déclencher l'écriture sur le disque seulement lorsque des latences inattendues surviennent. Notons toutefois que si les conditions sont mal définies, le portrait des latences dans le système observé sera incomplet. Une solution alternative serait de sélectionner aléatoirement des requêtes à tracer. Dans le cas de requêtes distribuées, il faudrait propager la

décision de tracer une requête d'une machine à l'autre, à la manière de Dapper. Notons que Dapper fonctionne uniquement en espace utilisateur. Propager cette décision dans le noyau du système d'exploitation serait un défi supplémentaire.

L'interface graphique de TraceCompare charge les caractéristiques de performance des exécutions à comparer en mémoire principale. Grâce aux ECCTs qui permettent de représenter ces caractéristiques de façon compacte, nous avons aisément pu comparer des groupes de centaines de milliers d'exécutions. La quantité de données générées par un système en production au fil du temps pourrait toutefois excéder la taille de la mémoire principale, ou même celle d'un disque dur. Pour pouvoir utiliser TraceCompare dans un tel cas, on pourrait concevoir un client léger déléguant l'exécution des algorithmes de comparaison à un système distribué.

6.3 Améliorations futures

Nous avons déjà discuté d'améliorations futures pour TraceCompare à la section 4.7. Nous ajoutons ici quelques idées supplémentaires.

La capture d'une pile d'appels en l'absence de *frame pointer* requiert des règles pour restaurer les valeurs des registres à chaque étage de la pile. Ces règles sont obtenues en exécutant du code à octets contenu dans la section `.eh_frame` des binaires ELF, une opération trop risquée pour être effectuée dans le noyau du système d'exploitation. Pour cette raison, nous devons envoyer un coûteux signal du noyau vers l'espace utilisateur à chaque fois que nous générons un événement `syscall_stack`. Nous avons vu que les règles de restauration des registres pouvaient être mises en cache une fois extraites de la section `.eh_frame`. Il serait intéressant de permettre au noyau d'accéder à cette cache. Ainsi, il ne serait pas nécessaire d'envoyer un signal lorsque les règles de restauration des registres sont présentes dans la cache pour tous les étages de la pile.

TraceCompare demande un effort de la part de l'utilisateur pour spécifier des groupes d'exécutions à comparer. Il serait intéressant d'intégrer à notre outil des algorithmes d'apprentissage machine pour former automatiquement des groupes d'exécutions similaires. En plus de simplifier l'utilisation de l'interface graphique, de tels algorithmes pourraient repérer automatiquement l'apparition de nouveaux groupes au fil du temps. Des alertes pourraient alors être envoyées aux personnes responsables du code ayant provoqué des variations de performance. Le mode *live* de LTTng pourrait être mis à profit pour générer ces alertes aussitôt que les variations se produisent.

La génération d'une base de données de ECCTs se fait en lisant séquentiellement une trace à partir d'un seul processeur. Cette opération pourrait être accélérée en lisant simultanément

différents segments de la trace à partir de différents processeurs. Un défi de cette parallélisation serait de gérer correctement les frontières entre les différents segments.

RÉFÉRENCES

[1] D. J. Dean, H. Nguyen, X. Gu, H. Zhang, J. Rhee, N. Arora, et G. Jiang, "Perfscope : Practical online server performance bug inference in production cloud computing infrastructures," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SOCC '14. New York, NY, USA : ACM, 2014, pp. 8 :1–8 :13.

[2] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, et C. Shanbhag, "Dapper, a large-scale distributed systems tracing infrastructure," *Google research*, 2010.

[3] J. Ciancutti, "Four reasons we choose amazon's cloud as our computing platform," http://techblog.netflix.com/2010/12/four-reasons-we-choose-amazons-cloud-as.html, décembre 2010, consulté le 23 mars 2015.

[4] I. Ceaparu, J. Lazar, K. Bessiere, J. Robinson, et B. Shneiderman, "Determining causes and severity of end-user frustration," *International Journal of Human-Computer Interaction*, vol. 17, no. 3, pp. 333–356, 2004.

[5] S. Work, "How loading time affects your bottom line," https://blog.kissmetrics.com/loading-time/, consulté le 23 mars 2015.

[6] M. Woodside, G. Franks, et D. Petriu, "The future of software performance engineering," in *Future of Software Engineering, 2007. FOSE '07*, May 2007, pp. 171–187.

[7] B. Gregg, "Freebsd flame graphs," http://www.brendangregg.com/blog/2015-03-10/freebsd-flame-graphs.html, mars 2015, consulté le 23 mars 2015.

[8] Y. Xiao, "Node.js in flames," http://techblog.netflix.com/2014/11/nodejs-in-flames.html, novembre 2014, consulté le 23 mars 2015.

[9] M. Desnoyers et M. R. Dagenais, "The lttng tracer : A low impact performance and behavior monitor for gnu/linux," in *Ottawa Linux Symposium*, Ottawa, Ontario, 2006, pp. 209–224.

[10] J. Desfossez et F. Doray, "Full stack system call latency profiling," https://lttng.org/blog/2015/03/18/full-stack-latencies/, mars 2015, consulté le 19 mars 2015.

[11] S. Rostedt, "Using the trace_event() macro," http://lwn.net/Articles/379903/, mars 2010, consulté le 26 janvier 2015.

[12] P.-M. Fournier, M. Desnoyers, et M. R. Dagenais, "Combined tracing of the kernel and applications with lttng," in *Ottawa Linux Symposium*, Ottawa, Ontario, 2009.

[13] C. Caldato, “Ms open tech contributes support for windows etw and perf counters to node.js,” http://blogs.msdn.com/b/interoperability/archive/2012/12/03/ms-open-tech-contributes-support-for-windows-etw-and-perf-counters-to-node-js.aspx, décembre 2012, consulté le 26 janvier 2015.

[14] B. Dawson, “Introduction to profiling with event tracing for windows,” https://www.wintellectnow.com/Videos/Watch/introduction-to-profiling-with-event-tracing-for-windows?videoId=introduction-to-profiling-with-event-tracing-for-windows, août 2014, consulté le 23 mars 2015.

[15] I. Park et R. Buch, “Improve debugging and performance tuning with etw,” https://msdn.microsoft.com/magazine/cc163437.aspx, avril 2007, consulté le 26 janvier 2015.

[16] B. Cantrill, M. W. Shapiro, A. H. Leventhal *et al.*, “Dynamic instrumentation of production systems.” in *USENIX Annual Technical Conference, General Track*, 2004, pp. 15–28.

[17] B. Gregg et J. Mauro, *DTrace : dynamic tracing in Oracle Solaris, Mac OS X, and FreeBSD*. Upper Saddle River, NJ : Prentice Hall, 2011.

[18] M. Desnoyers, “Low-impact operating system tracing,” Thèse de doctorat, École Polytechnique de Montréal, décembre 2009.

[19] Y. Brosseau, “A userspace tracing comparison : Dtrace vs lttng ust,” http://www.dorsal.polymtl.ca/fr/blog/yannick-brosseau/userspace-tracing-comparison-dtrace-vs-lttng-ust, décembre 2011, consulté le 19 mars 2015.

[20] IMP developpers, “Profiling vs. tracing,” http://ipm-hpc.sourceforge.net/profilingvstracing.html, septembre 2009, consulté le 2 février 2015.

[21] R. A. Vitillo, “Performance tools developments,” http://indico.cern.ch/event/141309/session/4/contribution/20/material/slides/0.pdf, juin 2011, consulté le 2 février 2015.

[22] J. Olsa, “perf : Add backtrace post dwarf unwind,” http://lwn.net/Articles/499116/, mai 2012, consulté le 19 mars 2015.

[23] U. Fässler et A. Nowak, “perf file format,” https://openlab-mu-internal.web.cern.ch/openlab-mu-internal/03_Documents/3_Technical_Documents/Technical_Reports/2011/Urs_Fassler_report.pdf, septembre 2011, consulté le 2 février 2015.

[24] F. Eigler, “Sytemtap dtrace comparison,” https://sourceware.org/systemtap/wiki/SystemtapDtraceComparison, février 2015, consulté le 19 mars 2015.

[25] R. Haas, “perf : the good, the bad, the ugly,” http://rhaas.blogspot.ca/2012/06/perf-good-bad-ugly.html, juin 2012, consulté le 2 février 2015.

[26] S. Ghemawat, “Google perftools cpu profile,” http://google-perftools.googlecode.com/svn/trunk/doc/cpuprofile.html, juin 2010, consulté le 26 janvier 2015.

[27] ——, “Google perftools heap profile,” http://google-perftools.googlecode.com/svn/trunk/doc/heapprofile.html, juillet 2011, consulté le 2 février 2015.

[28] J. Sedlacek et T. Hurka, “Visualvm : Profiling applications,” http://visualvm.java.net/profiler.html, consulté le 2 février 2015.

[29] A. Montplaisir, N. Ezzati-Jivan, F. Wininger, et M. Dagenais, “Efficient model to query and visualize the system states extracted from trace data,” in *Runtime Verification*, ser. Lecture Notes in Computer Science, A. Legay et S. Bensalem, édit. Springer Berlin Heidelberg, 2013, vol. 8174, pp. 219–234.

[30] N. Ezzati-Jivan et M. R. Dagenais, “A stateful approach to generate synthetic events from kernel traces,” *Advances in Software Engineering*, vol. 2012, janvier 2012.

[31] F. Wininger, “Conception flexible d’analyses issues d’une trace système,” Mémoire de maîtrise, École Polytechnique de Montréal, avril 2014.

[32] F. Giraldeau, “How linux changes the cpu frequency,” http://multivax.blogspot.ca/2014/11/how-linux-changes-cpu-frequency.html, novembre 2014, consulté le 19 mars 2015.

[33] N. Ezzati-Jivan et M. R. Dagenais, “Multilevel visualization of large execution traces,” *Elsevier Journal of Visual Languages and Computing*, à paraître.

[34] A. Montplaisir-Gonçalves, N. Ezzati-Jivan, F. Wininger, et M. R. Dagenais, “State history tree : an incremental disk-based data structure for very large interval data,” in *International Conference on Social Computing*. IEEE, septembre 2013, pp. 716–724.

[35] J. Oskarsson, “Zipkin : a distributed tracing framework,” février 2012, Strange Loop. [en ligne]. Disponible sur : http://www.infoq.com/presentations/Zipkin

[36] R. Isaacs, P. Barham, J. Bulpin, R. Mortier, et D. Narayanan, “Request extraction in magpie : Events, schemas and temporal joins,” in *11th Workshop on ACM SIGOPS European Workshop*, New York, NY, 2004.

[37] R. Isaacs, P. Barham, R. Mortier, et D. Narayanan, “Magpie : Online modelling and performance-aware systems.” in *9th Workshop on Hot Topics in Operating Systems*, mai 2003, pp. 85–90.

[38] F. Giraldeau et M. R. Dagenais, “Approximation of critical path using low-level system events,” à paraître.

[39] P. E. McKenney, “Differential profiling,” in *Proceedings of the Third International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, 1995 (MASCOTS’95)*. IEEE, 1995, pp. 237–241.

[40] M. Schulz et B. de Supinski, “Practical differential profiling,” in *Euro-Par 2007 Parallel Processing*, ser. Lecture Notes in Computer Science, A.-M. Kermarrec, L. Bougé, et T. Priol, édit. Springer Berlin Heidelberg, 2007, vol. 4641, pp. 97–106.

[41] D. Lee, S. Cha, et A. Lee, “A performance anomaly detection and analysis framework for dbms development,” *Knowledge and Data Engineering, IEEE Transactions on*, vol. 24, no. 8, pp. 1345–1360, août 2012.

[42] C. Bezemer, E. Milon, A. Zaidman, et J. Pouwelse, “Detecting and analyzing i/o performance regressions,” *Journal of Software : Evolution and Process*, vol. 26, no. 12, pp. 1193–1212, 2014.

[43] S. Savari et C. Young, “Comparing and combining profiles,” in *Proceedings of the Second Workshop on Feedback-Directed Optimization (FDO)*. ACM, 2000, pp. 50–62.

[44] S. B. Needleman et C. D. Wunsch, “A general method applicable to the search for similarities in the amino acid sequence of two proteins,” *Journal of molecular biology*, vol. 48, no. 3, pp. 443–453, 1970.

[45] J. D. Thompson, D. G. Higgins, et T. J. Gibson, “Clustal w : improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice,” *Nucleic Acids Research*, vol. 22, no. 22, pp. 4673–4680, 1994.

[46] E. L. Sonnhammer et R. Durbin, “A dot-matrix program with dynamic threshold control suited for genomic dna and protein sequence analysis,” *Gene*, vol. 167, no. 1, pp. GC1–GC10, 1995.

[47] H. Li et N. Homer, “A survey of sequence alignment algorithms for next-generation sequencing,” *Briefings in Bioinformatics*, vol. 11, no. 5, pp. 473–483, 2010.

[48] M. Weber, R. Brendel, et H. Brunst, “Trace file comparison with a hierarchical sequence alignment algorithm,” in *IEEE 10th International Symposium on Parallel and Distributed Processing with Applications (ISPA)*, Leganés, Madrid, Espagne, juillet 2012, pp. 247–254.

[49] M. Idris, A. Mehrabian, A. Hamou-Lhadj, et R. Khoury, “Pattern-based trace correlation technique to compare software versions,” in *Autonomous and Intelligent Systems*, ser. Lecture Notes in Computer Science, M. Kamel, F. Karray, et H. Hagras, édit. Springer Berlin Heidelberg, 2012, pp. 159–166.

[50] X. Zhuang, S. Kim, M. i. Serrano, et J.-D. Choi, “Perfdiff : A framework for performance difference analysis in a virtual machine environment,” in *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. New York, NY : ACM, 2008, pp. 4–13.

[51] C. H. Kim, J. Rhee, H. Zhang, N. Arora, G. Jiang, X. Zhang, et D. Xu, "Introperf : Transparent context-sensitive multi-layer performance inference using system stack traces," *SIGMETRICS Performance Evaluation Review*, vol. 42, no. 1, pp. 235–247, juin 2014.

[52] J. Trumper, J. Dollner, et A. Telea, "Multiscale visual comparison of execution traces," in *IEEE 21st International Conference on Program Comprehension (ICPC)*, mai 2013, pp. 53–62.

[53] Barcelona Supercomputing Center, "Paraver : Comparing mpi vs mlp," http://www.bsc.es/computer-sciences/performance-tools/examples-analysis/comparing-mlp-and-mpi, consulté le 19 janvier 2015.

[54] X. Zhang et R. Gupta, "Matching execution histories of program versions," in *Proceedings of the 10th European Software Engineering Conference*. New York, NY : ACM, 2005, pp. 197–206.

[55] N. Johnson, J. Caballero, K. Chen, S. McCamant, P. Poosankam, D. Reynaud, et D. Song, "Differential slicing : Identifying causal execution differences for security applications," in *2011 IEEE Symposium on Security and Privacy (SP)*, May 2011, pp. 347–362.

[56] A. Hamou-Lhadj, S. S. Murtaza, W. Fadel, A. Mehrabian, M. Couture, et R. Khoury, "Software behaviour correlation in a redundant and diverse environment using the concept of trace abstraction," in *Proceedings of the 2013 Research in Adaptive and Convergent Systems*, ser. RACS '13. New York, NY : ACM, 2013, pp. 328–335.

[57] F. Giraldeau, "Why malware detection based on syscalls n-grams is unlikely to work," http://multivax.blogspot.ca/2015/04/why-malware-detection-based-on-syscalls.html, avril 2015, (Consulté le 28 juillet 2015).

[58] C. Parampalli, R. Sekar, et R. Johnson, "A practical mimicry attack against powerful system-call monitors," in *Proceedings of the 2008 ACM Symposium on Information, Computer and Communications Security*, ser. ASIACCS '08. New York, NY, USA : ACM, 2008, pp. 156–167.

[59] F. Langer et E. Oswald, "Using reference traces for validation of communication in embedded systems," in *Proceedings of The Ninth International Conference on Systems (ICONS)*. IARIA, 2014, pp. 203–208.

[60] R. R. Sambasivan, A. X. Zheng, E. Thereska, et G. R. Ganger, "Categorizing and differencing system behaviours," *Hot Topics in Autonomic Computing*, juin 2007.

[61] F. Giraldeau, J. Desfossez, D. Goulet, M. Dagenais, et M. Desnoyers, "Recovering system metrics from kernel trace," in *Ottawa Linux Symposium*, vol. 109, Ottawa, Ontario, 2011.

[62] B. Gregg, *Systems performance : enterprise and the cloud*. Upper Saddle River, NJ : Prentice Hall, 2014.

[63] P. Reynolds, C. E. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, et A. Vahdat, "Pip : Detecting the unexpected in distributed systems." in *Proceedings of the 3rd Symposiun on Networked Systems Design and Implementation (NSDI)*. USENIX Association, mai 2006.

[64] W. A. Shewhart, *Economic control of quality of manufactured product*. ASQ Quality Press, 1931, vol. 509.

[65] T. H. Nguyen, B. Adams, Z. M. Jiang, A. E. Hassan, M. Nasser, et P. Flora, "Automated detection of performance regressions using statistical process control techniques," in *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '12. New York, NY : ACM, 2012, pp. 299–310.

[66] M. Munawar, T. Reidemeister, M. Jiang, A. George, et P. Ward, "Adaptive monitoring with dynamic differential tracing-based diagnosis," in *Managing Large-Scale Service Deployment*, ser. Lecture Notes in Computer Science, F. De Turck, W. Kellerer, et G. Kormentzas, édit. Springer Berlin Heidelberg, 2008, vol. 5273, pp. 162–175.

[67] B. Gregg, "Frequency trails," http://www.brendangregg.com/frequencytrails.html, février 2014, consulté le 23 janvier 2015.

[68] G. Jiang, H. Chen, et K. Yoshihira, "Discovering likely invariants of distributed transaction systems for autonomic system management," in *IEEE International Conference on Autonomic Computing 2006 (ICAC)*. IEEE, June 2006, pp. 199–208.

[69] Z. Guo, G. Jiang, H. Chen, et K. Yoshihira, "Tracking probabilistic correlation of monitoring data for fault detection in complex systems," in *Dependable Systems and Networks, 2006. DSN 2006. International Conference on*, June 2006, pp. 259–268.

[70] G. Jiang, H. Chen, et K. Yoshihira, "Efficient and scalable algorithms for inferring likely invariants in distributed systems," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 19, no. 11, pp. 1508–1523, Nov 2007.

[71] B. Gregg, "From clouds to roots : Performance analysis at netflix," http://www.brendangregg.com/blog/2014-09-27/from-clouds-to-roots.html, septembre 2014, consulté le 24 janvier 2015.

[72] N. A. James, A. Kejariwal, et D. S. Matteson, "Leveraging Cloud Data to Mitigate User Experience from "Breaking Bad"," *ArXiv e-prints*, novembre 2014.

[73] A. Kejariwal, "Introducing practical and robust anomaly detection in a time series," https://blog.twitter.com/2015/introducing-practical-and-robust-anomaly-detection-in-a-time-series, janvier 2015, consulté le 26 janvier 2015.

[74] A. Stanway, “Introducing kale,” https://codeascraft.com/2013/06/11/introducing-kale/, juin 2013, consulté le 20 mars 2015.

[75] E. Nygren, R. K. Sitaraman, et J. Sun, “The akamai network : A platform for high-performance internet applications,” *SIGOPS Operating Systems Review*, vol. 44, no. 3, pp. 2–19, Aug. 2010.

[76] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, et N. R. Tallent, “Hpctoolkit : tools for performance analysis of optimized parallel programs,” *Concurrency and Computation : Practice and Experience*, vol. 22, no. 6, pp. 685–701, 2010.

[77] N. Ezzati-Jivan et M. R. Dagenais, “A framework to compute statistics of system parameters from very large trace files,” *SIGOPS Operating Systems Review*, vol. 47, no. 1, pp. 43–54, janvier 2013.

[78] The Chromium Authors, “Performance profiling with the timeline,” https://developer.chrome.com/devtools/docs/timeline, s.d., consulté le 24 mars 2015.

[79] F. Rajotte et M. R. Dagenais, “Real-time linux analysis using low-impact tracer,” *Advances in Computer Engineering*, vol. 2014, juin 2014.

[80] B. Gregg, “Differential flame graphs,” http://www.brendangregg.com/blog/2014-11-09/differential-flame-graphs.html, novembre 2014, consulté le 24 mars 2015.

[81] N. Joukov, A. Traeger, R. Iyer, C. P. Wright, et E. Zadok, “Operating system profiling via latency analysis,” in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, ser. OSDI ’06. Berkeley, CA, USA : USENIX Association, 2006, pp. 89–102.

[82] kernel.org, “perf : Linux profiling with performance counters,” https://perf.wiki.kernel.org/, juin 2014, consulté le 23 avril 2015.

[83] J. Oakley et S. Bratus, “Exploiting the hard-working dwarf : Trojan and exploit techniques with no native executable code,” in *Proceedings of the 5th USENIX Conference on Offensive Technologies*, ser. WOOT’11. Berkeley, CA, USA : USENIX Association, 2011.

[84] F. Nybäck, “Improving the support for arm in the igprof profiler,” Mémoire de maîtrise, Aalto University, octobre 2014.

[85] G. Ammons, T. Ball, et J. R. Larus, “Exploiting hardware performance counters with flow and context sensitive profiling,” *SIGPLAN Notices*, vol. 32, no. 5, pp. 85–96, mai 1997.

[86] E. Merlo et T. Lavoie, “Computing structural types of clone syntactic blocks,” in *Reverse Engineering, 2009. WCRE ’09. 16th Working Conference on*, Oct 2009, pp. 274–278.

[87] N. R. Tallent, J. M. Mellor-Crummey, et M. W. Fagan, “Binary analysis for measurement and attribution of program performance,” *SIGPLAN Notices*, vol. 44, no. 6, pp. 441–452, juin 2009.

[88] T. Newton, “Two minute drill : Introduction to xperf,” http://blogs.technet.com/b/askperf/archive/2008/06/27/an-intro-to-xperf.aspx, juin 2008, (Consulté le 1 mai 2015).

ANNEXE A SCRIPT POUR CAPTURER LES ÉVÉNEMENTS REQUIS PAR TRACECOMPARE AVEC DTRACE

```
#!/usr/sbin/dtrace -s

#pragma D option quiet
#pragma D option ustackframes=100
#pragma D option stackframes=100

#pragma D option aggsize=8m
#pragma D option bufsize=16m
#pragma D option dynvarsize=16m

/* Stack events */
profile-199 /execname == "prime"/ {
        printf("[cpu-stack] pid=%d, ts=%u",
                pid, timestamp);
        ustack();
}

sched:::sleep /execname == "prime"/ {
        self->sleep_ts = timestamp;
}

sched:::on-cpu /self->sleep_ts &&
                timestamp - self->sleep_ts > 100000 / {
        printf("[sleep-stack] pid=%d, ts=%u, dur=%u",
                pid, timestamp, timestamp - self->sleep_ts);
        ustack();
    self->sleep_ts = 0;
}

/* Critical path events */
sched:::on-cpu {
        printf("[oncpu] prev_pid=%d, next_pid=%d, ts=%u, dur=%u\n",
```

```
                pid, args[0]->pr_lwpid, timestamp,
                timestamp - self->off_ts);
}

sched:::wakeup {
        printf("[wakeup] src=%d, target=%d, ts=%u\n",
                pid, args[0]->pr_lwpid, timestamp);
}

tcp:::send {
        printf("[send] pid=%d, seq=%d, flags=%d, ts=%u\n",
                pid, args[4]->tcp_seq, args[4]->tcp_flags, timestamp);
}

tcp:::receive {
        printf("[recv] pid=%d, seq=%d, flags=%d, ts=%u\n",
                pid, args[4]->tcp_seq, args[4]->tcp_flags, timestamp);
}
```